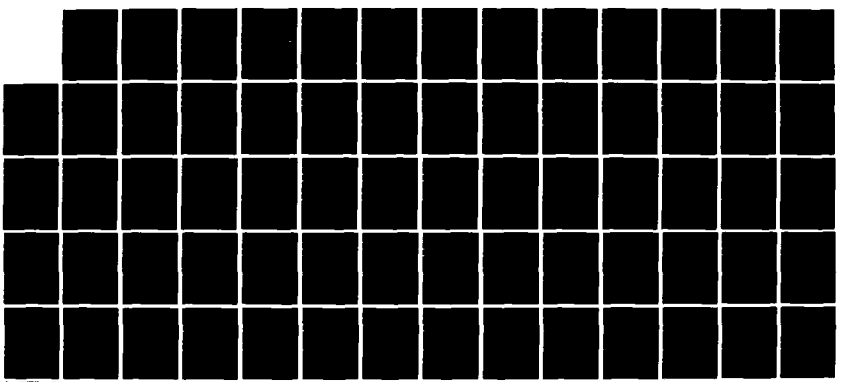
AD-A122 108 LETS: AN EXPRESSIONAL LOOP NOTATION(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB R C WATERS OCT 82 AI-M-680 N00014-80-C-0505 1/1

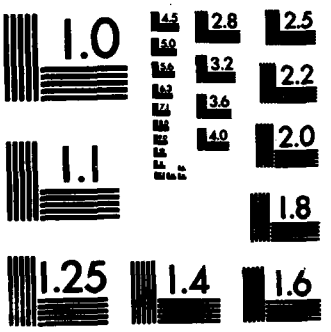
UNCLASSIFIED F/G 9/2 NL



END

FILMED

DTIC



(12)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: MEMO 680	2. GOVT ACCESSION NO.: AD-A122108
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): LETS: An Expressional Loop Notation	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Richard C. Waters	8. CONTRACT OR GRANT NUMBER(s): N00014-80-C-0505 MCS-7912179
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209	12. REPORT DATE: October 1982
	13. NUMBER OF PAGES: pages 65
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research Information Systems Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Loops
Programming Languages
Lisp

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Many loops can be more easily understood and manipulated if they are viewed as being built up out of operations on sequences of values. A notation is introduced which makes this viewpoint explicit. Using it, loops can be represented as compositions of functions operating on sequences of values. A library of standard sequence functions is provided along with facilities for defining additional ones.

CONTINUED NEXT PA(

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The notation is not intended to be applicabel to every kind of loop. Rather, it has been simplified wherever possible so that straightforward loops can be represented extremely easily. The expressional form of the notation makes it possible to construct and modify such loops rapidly and accurately. The implementation of the notatio does not actually use sequences but rather compiles loop expressions into iterative loop code. As a result, using the notation does not lead to a reduction in run time efficiency.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 680 October 1982

LetS

An Expressional Loop Notation

by

Richard C. Waters

ABSTRACT

Many loops can be more easily understood and manipulated if they are viewed as being built up out of operations on sequences of values. A notation is introduced which makes this viewpoint explicit. Using it, loops can be represented as compositions of functions operating on sequences of values. A library of standard sequence functions is provided along with facilities for defining additional ones.

The notation is not intended to be applicable to every kind of loop. Rather, it has been simplified wherever possible so that straightforward loops can be represented extremely easily. The expressional form of the notation makes it possible to construct and modify such loops rapidly and accurately. The implementation of the notation does not actually use sequences but rather compiles loop expressions into iterative loop code. As a result, using the notation does not lead to a reduction in run time efficiency.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, and in part by National Science Foundation grant MCS-7912179.

Introduction

This paper presents an expressional loop notation based on the ideas described in [16,17]. The notation makes it possible to represent loops as compositions of functions applied to sequences of values. The principal benefit of the notation is that it brings the powerful metaphor of expressions and decomposability to bear on the domain of loops. Wherever this metaphor can be applied, it makes algorithms much easier to construct, understand, and modify.

The paper is divided into four parts. The first part discusses what it means to view a loop as an expression composed of functions operating on sequences of values. It then presents the major features of the notation in terms of the expressional metaphor. It concludes by discussing the key places where the notation does not completely support the expressional metaphor.

The implementation of the notation does not support sequences as actual data objects, but rather compiles loop expressions into iterative loops which operate on sequences one element at a time. The second section of the paper presents a number of additional features of the notation which are best understood from the point of view of this element at a time perspective. This part of the paper concludes with a large example which shows the way the notation is intended to be used.

The third section of the paper evaluates the notation from several points of view. First, the limits of the applicability of the notation are described in detail. The notation is not intended to be applicable to every kind of loop. Rather, it is designed to make it particularly easy to represent and manipulate the kind of straightforward loops which appear most commonly in programs. By focusing on the main concept and resisting the temptation to add embellishments, the notation is rendered semantically clean and easy to understand.

Second, the efficiency of the code produced for loop expressions is discussed. Due to the fact that the notation can be directly compiled into iterative loop code, there is no need to suffer the kind of efficiency penalties which would be associated with actually implementing the notation in terms of data objects representing sequences. Appendix A contains an in depth description of the compilation process.

Third, it is argued that the notation could be implemented as a logical extension to almost any language. The notation has already been implemented as a LispMachine [18]/MacLisp [9] macro package LETS ("*let ess*"). (Note that several of the macros described in this paper end in the letter "S". This "S" stands for "sequence", and in all cases it is pronounced separately.) This paper discusses the notation in the context of this particular implementation and the examples are all couched in terms of Lisp. However, none of the basic concepts behind the notation have anything to do with the Lisp language *per se*. Introducing the expressional notation as an extension to the language Ada [1] is discussed.

The fourth and final part of the paper presents a comprehensive comparison between the expressional notation and other looping constructs. The concept of expressional loops presented here was motivated by observing regularities in the kinds of straightforward loops which appear in programs most often [16]. Over the years, many language designers have also noticed various aspects of these regularities and therefore many of the key features of the expressional notation appear in one form or another in currently existing looping constructs. The constructs which are most similar appear in the languages APL [10], Hibol [13], and Model [11]. The advantage of the notation presented here is that it distills these concepts into a semantically complete whole which is easy to understand, easy to compile, and easy to add as an extension to current languages.

1 - The Expressional Metaphor

The key property of expressions which makes them particularly easy to construct, manipulate, and understand is *decomposability*. Given an expression, it is easy to decompose it into separate parts each of which (in the absence of side-effects) can be completely understood in isolation from all of the other parts. Further, the behavior of the expression as a whole is merely the composition of the behaviors of its parts.

Consider the expression "`(SIN (SQRT X))`". Its two parts can be understood in isolation. For example, you can understand what the `SQRT` does (i.e., compute the square root of its input) without having to think about where its input comes from, where its output will be used, or about anything else that is going on in the expression. The only interaction between the two functions is the data flow between them. In order to understand what the expression as a whole does, (i.e., compute the sine of the square root of its input) you merely have to compose your understandings of the two functions.

The primary goal behind the design of the loop notation presented here has been the development of a notation which has the property of decomposability.

Viewing Loops as Expressions Involving Sequences

In order to represent loops as expressions, the concepts of *sequences* and *sequence functions* which operate on them are introduced. In this context, all other data structures are referred to as *unitary*. A sequence is an ordered (possibly infinite) one dimensional series of unitary data objects. A sequence function is a function which produces one or more sequences as outputs and/or consumes one or more sequences as inputs. Loops are represented as expressions built out of sequence function applications.

For reasons of efficiency, sequences are not represented as actual data structures at run time. Rather, expressions involving sequences are compiled into iterative loops in which the existence of the sequences is only implicit. This is analogous to the way in which many program constructs are handled by compilers. For example, references to components of a record structure in a program typically appear to pass indirectly through the structure as a whole. However, for efficiency, such references are generally compiled into direct accesses on the components as if they were atomic objects. The existence of the structure as an identifiable unit is only implicit in the compiled code.

Sequences and sequence functions exist as explanatory devices. The point is that thinking of loops as compositions of functions operating on sequences makes them easier to understand. The fact that the compiled form is very different is in general of no import. (The second part of this paper discusses situations where the user does have to be cognizant of the compiled form.)

Consider the program `SUM-POSITIVE-EXPRESSIONAL` below. Its body is a sequence expression which sums up the positive elements of a one dimensional array. Given an array containing `<0 1 -1 2 -2>` the program would produce the result 3.

```
(defun sum-positive-expressional (vector)
  (Rsum (Fgreater (Evector vector))))
```

The sequence function `EVECTOR` ("*ee vector*") takes in a one dimensional array and enumerates a sequence of the data items in the array (e.g., producing the sequence `[0 1 -1 2 -2]`). (Note that the names of the built-in sequence functions all begin with prefix letters. These letters indicate the type of operation performed by the sequence function. The letter "`E`" stands for enumerate, "`G`" stands for generator, "`F`" stands for filter, and "`R`" stands for reduce. In each case, these prefix letters are pronounced separately.)

The sequence function `FGREATER` ("*ef greater*") takes in a sequence and filters it producing a sequence containing only the positive elements in the input sequence (e.g., producing `[_ 1 _ 2 _]`). Note that the action of the filter is encoded by leaving some of the slots in the output sequence empty (symbolized by "`_`")

rather than by creating a sequence of reduced length. In order to make this work, everything is set up so that empty slots are ignored in subsequent computations. The reason why the concept of empty slots is useful stems from the element at a time metaphor and will be discussed in the second part of this paper.

The sequence function RSUM ("*ar sum*") takes in a sequence of integers and reduces it to a unitary object containing their sum (e.g., 3). The sequence expression above is easy to understand because the actions of the sequence functions can be understood in isolation from each other, and the action of the expression as a whole (i.e., to sum the positive elements of a vector) is simply the composition of these actions. Further, it is as easy to modify as any other expression.

Simple Examples of Sequence Functions

This section presents a number of built-in sequence functions which are used in examples in the rest of this paper. The complete set of built-in sequence functions provided as part of the LETS macro package is presented in Appendix B. There are three basic kinds of sequence functions: *unitary→sequence*, *sequence→unitary*, and *sequence→sequence*. The most common kind of *unitary→sequence* function takes some aggregate data object and creates a sequence of its components.

Elist *list*

Takes in a list and creates a sequence of its elements.

e.g., `(Elist '(1 2 3)) => [1 2 3]`

Esublists *list*

Takes in a list and creates a sequence of its successive sublists.

e.g., `(Esublists '(1 2 3)) => [(1 2 3) (2 3) (3)]`

Evector *vector* **&optional** (*first* 0) (*last* (1- (array-length *vector*)))

Takes in a one dimensional array and creates a sequence of its elements.

e.g., `(Evector #<1 2 3>) => [1 2 3]`

Efile *file-name*

Creates a sequence of values by reading all of the objects out of the file.

e.g., `(Efile "data.lisp") => [1 2 3]`
`if the file "data.lisp" contains "1 2 3"`

Another family of *unitary→sequence* functions computes a sequence of values according to some formula.

Erange *first last* **&optional** (*step-size* 1)

Creates a sequence of integers by counting from *first* to *last* by the positive increment *step-size*.

e.g., `(Erange 4 8 2) => [4 6 8]`

Gsequence *object*

Generates an infinite sequence all of whose elements are *object*.

e.g., `(Gsequence 'A) => [A A A ...]`

The most common kind of *sequence→unitary* function takes in a sequence and combines the elements in it together into an aggregate data structure.

Rlist *sequence*

CONSes the non-empty values in a sequence into a list.

e.g., (Rlist [1 2 _ 3]) => (1 2 3)

Rvector *vector sequence* &optional (*first* 0) (*last* (1- (array-length *vector*)))

Stores the non-empty values in a sequence into successive slots of a one dimensional array.

e.g., (Rvector <A B C D> [1 2 _ 3]) => <1 2 3 D>

Rfile *file-name sequence*

Writes the non-empty values in a sequence into the indicated file.

e.g., (Rfile "data.lisp" [1 2 _ 3]) => T

"<cr>1 <cr>2 <cr>3 " is printed in "data.lisp"

Another kind of *sequence→unitary* function computes some summary value based on the values in the sequence.

Rsum *sequence-of-integers*

Computes the sum of the non-empty integer values in a sequence.

e.g., (Rsum [1 2 _ 3]) => 6

Rcount *sequence*

Counts the number of non-empty items in a sequence.

e.g., (Rcount [A B _ C]) => 3

Rlast *sequence* &optional (*default* NIL)

Returns the last non-empty element (if any) of the sequence as its value; otherwise returns *default*.

e.g., (Rlast [A B C _]) => C

Sequence→sequence functions take in a sequence of values and compute some related sequence. They tend to be much more idiosyncratic than other kinds of sequence functions and only one is predefined. The next section describes, among other things, the mechanisms which are used to create user defined *sequence→sequence* functions.

Fgreater *sequence-of-numbers* &optional (*limit* 0)

Selects the non-empty elements of a sequence of integers greater than *limit*.

e.g., (Fgreater [1 2 _ 3] 2) => [_ _ _ 3]

The programs below give a number of examples of loops built up out of the sequence functions described above. COPY-LIST copies a list by enumerating the items in the list and then CONSing them up into a new list. LAST enumerates all of the sublists in a list and then returns the last one. SUM-FIRST-N adds up the first N integers by enumerating the integers and then summing this sequence of values.

```
(defun copy-list (list)
  (Rlist (Elist list)))

(defun last (list)
  (Rlast (Esublists list)))

(defun sum-first-n (n)
  (Rsum (Erange 1 n)))
```

FILE-LENGTH computes the number of records in a file by enumerating them and then counting the items in this sequence. DUMP-VECTOR prints the elements of a vector into a file. ZERO-VECTOR initializes a vector by setting the elements to zero. It uses GSEQUENCE in order to generate a sequence of zeros to use.

```
(defun file-length (file-name)
  (Rcount (Efile file-name)))

(defun dump-vector (file-name vector)
  (Rfile file-name (Evector vector)))

(defun zero-vector (vector)
  (Rvector vector (Gsequence 0)))
```

Meta Sequence Functions

In addition to predefined sequence functions, the LETS macro package supports several *meta* sequence functions which make it easy for the user to create new sequence functions. The basic action of a meta sequence function is to take an ordinary function and convert it into a function on sequences. Each meta sequence function builds a particular kind of sequence function.

The most basic meta sequence function is (MAPS *function sequence ...*). MAPS is a generalization of the Lisp function MAP and is the principal method for creating user specified *sequence→sequence* functions. It takes *function* and converts it into a sequence function which takes in the sequence inputs and creates a sequence output. The number of sequences provided must be compatible with the number of arguments required by *function*. The nth element of the output sequence is computed by applying *function* to the nth elements of the input sequences. However, if the nth element of any of the input sequences is empty then *function* is not applied and the nth element of the output is empty. Note that the length of the output sequence is the same as the length of the shortest input sequence. The *function* parameter can be either a quoted function name, or a quoted LAMBDA expression (or a macro that expands into either one). For example, the program PAIRWISE-MAX takes in two lists and creates a list where each element is the maximum of the corresponding elements in the two input lists. The program SQUARE-LIST creates a list of the squares of the items in a list.

```
(defun pairwise-max (list1 list2)
  (Rlist (mapS #'max (Elist list1) (Elist list2))))

(defun square-list (list)
  (Rlist (mapS #'(lambda (x) (* x x)) (Elist list))))
```

The program TIMES-N multiplies every element in a list by a parameter N. The point of this example is that the functional argument to MAPS (and the functional arguments to the other meta sequence functions described below) can refer to any number of free variables. These free variables do not have to be declared special because the LETS macro package renders the loop entirely as inline code.

```
(defun times-n (list n)
  (Rlist (mapS #'(lambda (x) (* x n)) (Elist list))))
```

An extended form of MAPS is the meta sequence function (SCANS *function init sequence ...*). This creates a sequence function with an internal state variable. The input *function* must be a function of n+1 arguments where n is the number of sequences supplied. The elements of the output are the successive values of the state not including its initial (unitary) value *init*. The nth value of the state is computed by calling *function* with the prior value of the state as its first argument and the nth elements of the inputs as its remaining arguments. However, if the nth element of any of the input sequences is empty then *function* is not applied,

the state is not changed, and the nth element of the output is empty. As with MAPS, the length of the output sequence is the same as the length of the shortest input sequence. SCANS is useful for creating a sequence function corresponding to a recurrence relation. For example, the program SQUARES computes a list of the first N squares without doing any multiplication by taking advantage of fact that $n^2 = (n-1)^2+2n-1$.

```
(defun squares (n)
  (Rlist (scanS #'(lambda (n-1-squared n) (+ n-1-squared n n -1))
                0 (Erange 1 n))))
```

The meta sequence function (FILTERS *function sequence*...) is used to create sequence functions like FGREATER which select a subsequence of a sequence. The elements of the output sequence are computed as follows. If the result of applying *function* to the nth elements of the input sequences is non-NIL then the nth element of the *first* input is used as the nth element of the output; otherwise the nth output element is empty. As with MAPS, if the nth element of any of the input sequences is empty then *function* is not applied and the nth element of the output is empty.

As an example, consider the function SUM-POSITIVE-FILTER. It uses the meta sequence function FILTERS instead of the sequence function FGREATER. The function REMQ takes in a list and CONSes up a new list which is the same except that all instances of a given item are removed. It uses a filter to select which list elements to keep.

```
(defun sum-positive-filter (vector)
  (Rsum (filterS #'plusp (Evector vector))))

(defun remq (item list)
  (Rlist (filterS #'(lambda (x) (not (eq x item))) (Elist list))))
```

User specified *sequence→unitary* functions can be created by using the meta sequence function (REDUCES *function init sequence*...). This creates a sequence function with an internal state variable. The state is initialized to the (unitary) value *init*. The nth value of the state is computed by calling *function* with the prior value of the state as its first argument and the nth elements of the inputs as its remaining arguments. However, if the nth element of any of the input sequences is empty then *function* is not applied and the state is not changed. When the input sequences are exhausted, the final value of the state variable is returned as the (unitary) result. If there are no non-empty elements in the input sequences then the value *init* will be returned. The meta sequence functions REDUCES and SCANS are very closely related. The expression (REDUCES *function init sequence*) is the same as (RLAST (SCANS *function init sequence*) *init*).

As examples, consider the following two functions. SUM-POSITIVE-REDUCER uses REDUCES instead of a call on RSUM. MAKE-SET takes in a list possibly containing duplicate elements and creates a list without any duplicates which contains the same elements. The key problem is removing duplicates. To do this, the function uses a reducer which adds the current element into the list being created only if it is not a member of the list already.

```
(defun sum-positive-reducer (vector)
  (reduceS #'+ 0 (Fgreater (Evector vector))))

(defun make-set (list)
  (reduceS #'(lambda (state item)
               (cond ((member item state) state)
                     (T (cons item state))))
           nil
           (Elist list)))
```

The most basic way to create a *unitary→sequence* function is to use the meta sequence function (GENERATES *function init*). The sequence function produced creates a sequence of elements where the (unitary) value *init* is the first element and where each successive element is computed from the prior element by evaluating *function* with the prior element as its argument. Note that the output sequence is infinite in extent. The next two meta sequence functions can be used to create finite sequences.

A loop expression which contains only a generator will never terminate because it operates on an infinite sequence. However, if a loop expression is working on several sequences some of which are finite and some of which are not, it will terminate as soon as the shortest finite sequence has been exhausted. This is discussed further in the section on termination below.

Generators are typically used in loop expressions in conjunction with finite sequences of unknown length. For example, the program DIGITS-TO-NUMBER takes in a list of one digit numbers and computes the corresponding integer (e.g., '(1 2 3) becomes 123). The loop expression works with two basic sequences. It enumerates the digits in the list in reverse order (i.e., least significant digit first). It also creates an unbounded sequence of scale factors consisting of the successive powers of ten. The result is computed by summing up the product of each digit with its corresponding scale factor. The loop terminates when the digits run out.

```
(defun digits-to-number (digit-list)
  (Rsum (mapS #'* (Elist (reverse digit-list))
                  (generateS #'(lambda (x) (* x 10)) 1))))
```

Another example is the program FILL-VECTOR which takes in a list and uses it to initialize the elements of a vector. If there are more elements in the list than in the vector, the extra list elements are ignored. On the other hand, if there are more elements in the vector, the last element of the list is used to fill out all of the remaining elements of the vector.

```
(defun fill-vector (vector list)
  (Rvector vector (mapS #'car (generateS #'(lambda (x) (or (cdr x) x)) list))))
```

Note that it is the size of the vector which controls the computation, not the length of the list. To do this conveniently, a generator is created which generates the successive sublists of the list, but which continues to generate the last sublist indefinitely once it has been reached. The function CAR is MAPSed over the generated sequence in order to get the desired list elements. These elements are then stored in the vector. RVECTOR contains a termination which stops the loop when the vector has been filled up.

The meta sequence function (TRUNCATES *function sequence...*) is used to create sequence functions which take in potentially infinite sequences and return sequences which have been truncated to finite length. The *function* argument is applied to successive groups of corresponding elements of the input sequences. The output sequence is composed of the elements of the *first* input sequence up to but not including the first element corresponding to a non-NIL evaluation of *function*. As with the other meta sequence functions, if any of the nth elements of the input sequences are empty, then *function* is not applied and the nth output element is empty. Note that the output sequence is typically shorter than any of the input sequences, and can be of

length zero.

Consider the following two examples which are the same as programs presented in the last section except that they use explicit generators and truncators. SUM-FIRST-N-TRUNCATOR generates an infinite sequence of integers and truncates it after N. The resulting sequence is then summed. LAST-TRUNCATOR generates the successive CDRs of a list and truncates this when NIL is reached. The last sublist is returned.

```
(defun sum-first-n-truncator (n)
  (Rsum (truncateS #'(lambda (x) (> x n)) (generateS #'1+ 1))))

(defun last-truncator (list)
  (Rlast (truncateS #'null (generateS #'cdr list))))
```

Truncators are a great deal like filters in that they take in a sequence and return a restricted sequence. However, they differ in one very important way -- they cause the loop to terminate. Even when a filter selects only a finite number of elements out of an infinite sequence, it never causes the loop to terminate. For example, the program SUM-FIRST-N-BUGGY will never terminate even though the correct numbers have been selected. Note that it is not in general possible to detect when a filter has reached a point where no more elements of the input sequence will satisfy the filter test.

```
(defun sum-first-n-buggy (n)
  (Rsum (filterS #'(lambda (x) (not (> x n))) (generateS #'1+ 1))))
```

The meta sequence function (ENUMERATES *truncate-function generate-function init*) is an abbreviation for the common combination (TRUNCATES *truncate-function* (GENERATES *generate-function init*)). For example the two functions above could be written more compactly as follows:

```
(defun sum-first-n-enumerator (n)
  (Rsum (enumerateS #'(lambda (x) (> x n)) #'1+ 1)))

(defun last-enumerator (list)
  (Rlast (enumerateS #'null #'cdr list)))
```

The meta sequence functions are an essential part of the expressional loop notation because they provide a convenient mechanism whereby the user can create additional operations on sequences.

LetS

In an ordinary expression, if you want to use the value of a subexpression in two places, you have to bind this value to a variable. The prototypical way to do this in Lisp is with the macro LET. The macro LETS fills the identical role in sequence expressions. In the absence of side-effects, the only effect of using LET or LETS, rather than merely duplicating the subexpression, is increased efficiency due to executing the subexpression only once and a potential gain in the clarity of the expression.

The macro LETS is analogous to destructuring LET*. It has a list of bound variable value pairs which are executed sequentially so that you can use a variable in the computation of the value to be bound to a later variable. Instead of a single variable, a tree of variables can be used to specify destructuring. Alternately, the value can be omitted in which case it is assumed that there is no initial value at all. In this case the variable must be set before it can be read. These three cases are illustrated in the example below.

```
(letS ((v1 value1) ((v2 v3) value2) v4)
  . body)
```

Inside the body of a LETS you can use the form (SETQ *variable value*) in order to assign a sequence value to a sequence variable. The initializing values are handled as if they were sequentially assigned to the bound variables inside the LETS as illustrated below. Destructuring is implemented in terms of the appropriate CAR

d CDR operations.

```
(letS (v1 x v2 v3 v4)
  (setq v1 value1)
  (setq x value2)
  (setq v2 (mapS #'car x))
  (setq v3 (mapS #'cadr x))
  . body)
```

Each initializing value must be a sequence. LETS cannot be used to bind a variable to a unitary value. owever, using GSEQUENCE, you can bind a sequence variable to an infinite sequence of a unitary value, ich will usually be sufficient.)

The macro LETS contains a body which consists of one or more loop expressions. These expressions can er to the sequences bound to the sequence variables, and can result in either sequence or unitary values. e value of the last form must be unitary and is returned as the result of the LETS as a whole. (This and ne additional aspects of LETS stem from the element at a time metaphor and will not be discussed in detail til the next part of the paper.)

As a simple example, DIGITS-TO-NUMBER could be rewritten as shown below. Two sequence variables : used to make the loop more readable.

```
(defun digits-to-number-letS (digit-list)
  (letS ((digits (Elist (reverse digit-list)))
         (scales (generateS #'(lambda (x) (* x 10)) 1)))
    (Rsum (mapS #'* digits scales))))
```

Since each sequence variable is only used once in DIGITS-TO-NUMBER-LETS the function is orithmically identical to the earlier version of this function which did not use variables. A sequence iable can of course be referenced more than once. This will not cause the sequence to be computed more n once. As a result, SQUARE-LIST-LETS below is more efficient than SQUARE-LIST-REDUNDANT.

```
(defun square-list-letS (list)
  (letS ((integers (Elist list)))
    (Rlist (mapS #'* integers integers))))

(defun square-list-redundant (list)
  (Rlist (mapS #'* (Elist list) (Elist list))))
```

The explicit use of SETQ in a LETS is illustrated in the program DIGITS-TO-NUMBER-SETQS. The example ws that you can make repetitive assignments redefining the value of a sequence variable just as you can in ordinary LET. The program first enumerates the digits. It then multiplies each one by the appropriate le factor and then sums the resulting sequence. It is important to note that you cannot use anything other n SETQ (or, as discussed below, MULTIPLE-VALUE) in order to assign to a sequence variable. In particular i cannot use SETF or any other macro even if it expands into a SETQ.

```
(defun digits-to-number-setq (digits)
  (letS (integers)
    (setq integers (Elist (reverse digits)))
    (setq integers (* integers (generateS #'(lambda (x) (* x 10)) 1)))
    (Rsum integers)))
```

SETQs and other forms can be used to assign to any number of free (unitary) variables in the body of a S. This is often useful for passing information out of a loop.

DefunS

The meta sequence functions make it possible for a user to create his own sequence operations. However, these operations are only literals and must be recreated each time they are to be used. The macro DEFUNS makes it possible for a user to created *named* sequence functions which he can then use in loop expressions. These sequence functions are actually macros which are compiled inline by the LETS macro package. However, in the context of the expressional notation, they are intended to be thought of as functions just like any other function.

(defunS *name parameter-list*
. *body*)

The macro DEFUNS is exactly analogous to DEFUN. It has two basic parts: a parameter list and a body. The parameter list is a list of variable names and supports four keywords: &UNITARY, &SEQUENCE, &OPTIONAL, and &AUX. Each of the keywords is *sticky* and specifies the type of all of the parameters which follow it until another keyword changes the type. The first two keywords are used to specify whether a particular parameter is a sequence or an ordinary unitary object. By default, the parameters are initially assumed to be unitary. Just as in ordinary Lisp, &OPTIONAL specifies that the following parameters are optional. Also just as in ordinary Lisp, &AUX specifies that the following variables are not parameters at all, but rather just internal values. Optional/initial values can be specified using variable value pairs. However, unlike ordinary Lisp, if no default value is specified then no value will be supplied and the associated variable must be set before it can be read.

The body of a DEFUNS is exactly the same as the body of a LETS except that the last form is not required to yield a unitary value. Note, however, that DEFUNS is completely different from LETS in that it creates a sequence function which can later be combined together with other sequence functions while LETS creates an actual loop. The value of the last form, be it unitary or sequence, is returned as the value of the sequence function being created. The following examples use DEFUNS in order to define a number of the standard sequence functions described above. Note that ELIST returns a sequence while RSUM returns a unitary value.

```
(defunS Elist (list)
  (mapS #'car (enumerateS #'null #'cdr list)))

(defunS Rsum (&sequence num)
  (reduceS #'+ 0 num))
```

The following definitions illustrate the use of optional parameters. ERANGE takes an optional positive increment which defaults to one. FGREATER takes an optional comparison limit which defaults to zero. EVECTOR takes an optional interval in the vector which defaults to the full limits of the vector.

```
(defunS Erange (first last &optional (step-size 1))
  (enumerateS #'(lambda (x) (> x last)) #'(lambda (x) (+ x step-size)) first))

(defunS Fgreater (&sequence integers &unitary &optional (limit 0))
  (filterS #'(lambda (x) (> x limit)) integers))

(defunS Evector (vector &optional (start 0) (end (1- (array-length vector))))
  (mapS #'(lambda (x) (aref vector x)) (Erange start end)))
```

The ability for the user to conveniently define his own named sequence functions is a particularly important part of the expressional loop notation. It makes it possible for him to extend the notation to deal with the particular data abstractions he creates. A detailed example of this is given in a later section.

Multiple Values

All of the sequence functions presented above have had a single return value. The LETS macro package supports multiple return values by supporting the standard LispMachine functions VALUES and MULTIPLE-VALUE inside the body of a DEFUNS or LETS. For example, the function EPLIST takes in a disembodied plist and returns two values: a sequence of the property names, and a sequence of the values of those properties. VALUES is used as the last form of the DEFUNS in order to specify that two sequences are being returned.

```
(defunS Eplist (plist &sequence &aux pointers)
  (setq pointers (enumerateS #'null #'cddr (cdr plist)))
  (values (mapS #'car pointers) (mapS #'cadr pointers)))
```

The program PLIST-TO-ALIST converts a plist into an alist where the entries in the alist are created by CONSing together successive property value pairs. The program illustrates how MULTIPLE-VALUE can be used in a LETS in order to access the two sequences returned by EPLIST.

```
(defun plist-to-alist (plist)
  (letS (properties values)
    (multiple-value (properties values) (Eplist plist))
    (rlist (mapS #'cons properties values))))
```

The function COUNT-AND-SUM illustrates the use of VALUES as the last form of a LETS in order to specify that the loop as a whole returns multiple values.

```
(defun count-and-sum (list)
  (letS ((integers (Elist list)))
    (values (Rcount integers) (Rsum integers))))
```

(The MacLisp version of LETS does not support multiple return values from loops. However, it does support multiple return values from sequence functions such as EPLIST.)

Where the Expressional Metaphor Breaks Down

There are two situations in which loop expressions fail to be faithful to the expressional metaphor. The first of these involves side-effects. If a sequence function performs side-effects which disturb the actions of another sequence function, then the behavior of a loop expression as a whole can fail to be the composition of the behaviors of the sequence functions when looked at separately. (A detailed example of this will be discussed later on in this paper.) It should be noted that the breakdown of the expressional metaphor in this situation is not at all surprising considering that ordinary expressions lose the property of modularity in the presence of side-effects.

The second place where the expressional metaphor breaks down is when questions of termination are being considered. As mentioned above, the termination of a loop expression is controlled by the length of the sequences in it. The loop expression terminates as soon as the *shortest* sequence in it is exhausted. This is an example of *action at a distance* which makes it impossible to understand the various parts of a loop expression completely in isolation from each other.

Consider again the program DIGITS-TO-NUMBER (reproduced below). There are several questions about the sequence functions in this program which cannot be answered completely locally. For example, although it is convenient to describe the generator as creating an infinite sequence of powers of 10, it cannot actually do that. The generator will eventually halt with an error due to arithmetic overflow unless some other sequence function terminates the loop before then. On the other hand, in order to be sure that the ELIST will succeed in enumerating all of the digits, one must check that no other sequence function will terminate the loop before the end of the list of digits is reached. Because of these problems, you cannot just compose an

understanding of its parts in order to understand the loop expression as a whole.

```
(defun digits-to-number (digit-list)
  (Rsum (mapS #'* (Elist (reverse digit-list))
                  (generateS #'(lambda (x) (* x 10)) 1))))
```

Fortunately, there is a middle ground with regard to the property of decomposability. As discussed in [16], as long as you make no statements which depend on a specific minimum length of any sequence, any statement which is true about a sequence function in isolation will be true when it is composed with other functions in a loop expression. For example, you can say that the generator creates a sequence of powers of 10 beginning with 1. However, you cannot make any statement about whether it will or will not get arithmetic overflow in the process. Similarly, you can say that ELIST enumerates successive elements of a list starting with the first one. You can even say that it will produce a sequence no longer than the list. However, you cannot make any claim about any minimum number of list elements which definitely will be enumerated.

Given the kind of statements you can dependably make, you can determine a great deal about a loop by using straight composition. For example, in DIGITS-TO-NUMBER it is easy to tell that the values in the sequence created by the generator correspond to successive powers of 10, and that the sequence created by the ELIST correspond to successive digits least significant digit first. In addition, it is clear that the program multiplies each digit by the appropriate power of 10 and that these results are summed up.

In order to go beyond this and make statements about termination, you must do more global reasoning. In this case, there is only one basic finite sequence involved (the one created by the ELIST), and it clearly dominates the computation. As a result, the program clearly processes all of the digits and terminates computing the correct number.

The two step reasoning process outlined above is usually very satisfactory for the kind of straightforward loops the expressional notation is designed to represent. In particular, the global reasoning about termination is usually not at all difficult.

II - The Element at a Time Metaphor

The loop notation being presented here supports a second (element at a time) computational metaphor in addition to the expressional metaphor. The expressional metaphor is based on the idea that a sequence is a logical unit which is created in its entirety by one sequence function and then consumed by another sequence function. The element at a time metaphor is based on the idea that the computation involving all of the sequences in a single loop proceeds in parallel and that the loop expression is essentially describing what happens on a typical step in this process. For example, consider the following version of the program DIGITS-TO-NUMBER.

```
(defun digits-to-number-elements (digit-list)
  (letS ((digit (Elist (reverse digit-list)))
         (scale (generateS #'(lambda (x) (* x 10)) 1)))
    (Rsum (mapS #'* digit scale))))
```

In this program, the typical value of DIGIT is an element of the sequence created by enumerating the input DIGIT-LIST. The typical value of SCALE is a power of ten taken from the sequence created by the generator. On each cycle of the computation, the value of DIGIT is multiplied by the corresponding value of SCALE. The final result is the sum of these products.

If you compare DIGITS-TO-NUMBER-ELEMENTS with the program DIGITS-TO-NUMBER-LETS above you will see that the only actual difference is that the names of the sequence variables are singular instead of plural. The only important difference is in the way they are described. The element at a time description is very natural for some parts of the computation (e.g., the multiplication of corresponding values of DIGIT and SCALE). Other parts of the computation (e.g., the RSUM) only really make sense from the point of view of the expressional metaphor.

The element at a time metaphor is intimately tied up with the way loop expressions are actually compiled. The LETS macro package converts each loop expression into an ordinary iterative loop. This loop processes the sequences in the expression one element at a time with each slot in the sequences corresponding to one cycle of the loop which is produced. The correspondence between the loop produced and the element at a time metaphor is obvious. However, it should be noted that while the compilation process exists purely as a matter of efficiency, the element at a time metaphor is highlighted in this discussion because it is the motivation for a number of useful facilities supported by the expressional notation.

The two metaphors are really very separate ideas. One could easily decide to support just one of them. For example, the language APL supports much of the expressional metaphor and relatively little of the element at a time metaphor, while the language Hibol does the opposite. Experience with the expressional notation suggests that it is beneficial to blend these two ideas together. An interesting aspect of this is the fact that the restrictions on the expressional metaphor which are needed in order to clearly support the element at a time metaphor are essentially the same restrictions which are required in order to guarantee efficient compilation.

Implicit MapS

The creation of *sequence→sequence* functions with MAPS is so common that a syntactic sugaring has been introduced to make this easier. *Whenever an ordinary unitary Lisp function is applied to sequences, it is implicitly* MAPS*ed over those sequences.* For example, the following version of DIGITS-TO-NUMBER is equivalent to the one above.

```
(defun digits-to-number-implicit (digit-list)
  (letS ((digit (Elist (reverse digit-list)))
         (scale (generateS #'(lambda (x) (* x 10)) 1)))
    (Rsum (* digit scale))))
```

The function * in the last line of the program is applied to two sequence variables and its output is used as a sequence by the RSUM. From the point of view of the expressional metaphor this is a type conflict and does not make a great deal of sense. However, as a statement of what happens to the typical elements of the sequences in the variables DIGIT and SCALE (i.e., that they are multiplied) it makes a lot of sense. Implicit introduction of MAPS is provided in order to support this viewpoint.

MAPS is introduced implicitly in situations more complex than the one above. A full statement of the process is as follows: *If a unitary expression appears in an environment where a sequence value is expected then the entire expression down to, but not including, any components which create sequences is separated out as a* LAMBDA *expression and* MAPS*ed.* This is demonstrated by the following three pairs of equivalent loop expressions. The second pair illustrates the fact that an expression might not refer to any sequence values at all. In this case it will be converted to a LAMBDA expression of no arguments and still be MAPSed. (Note that, as is, this particular example would attempt to create an infinite list.) The third pair illustrates that implicit MAPS introduction is applied to expressions involving fexprs and macros in exactly the same way as it is to other expressions.

```
        (Rlist (1+ (* (Elist list1) (+ 2 (Elist list2)))))
same as: (Rlist (mapS #'(lambda (x y) (1+ (* x (+ 2 y))))
                      (Elist list1)
                      (Elist list2)))

        (Rlist (ncons T))
same as: (Rlist (mapS #'(lambda () (ncons T))))

        (Rlist (let ((z (Elist list)))
                 (push (* z z) stack)
                 z))
same as: (Rlist (mapS #'(lambda (x)
                          (let ((z x))
                            (push (* z z) stack)
                            z))
                      (Elist list)))
```

A potential aspect of the expressional metaphor is sacrificed in order to support implicit MAPS introduction. If nothing else were said you would expect that sequences were real data objects and that they could be passed between non-sequence functions and operated on by ordinary unitary functions like CONS. However, this is not possible because all such expressions are converted by implicit MAPS introduction into sequence functions. As a result, a sequence can never be operated on by anything other than a sequence function and sequences are always contained completely inside sequence expressions. It should be noted that this is a restriction which has to be imposed in any case in order to insure efficient compilation. If sequences were allowed to escape from the confines of sequence expressions then there would have to be an explicit representation for them, and it would not be possible to efficiently compile any loop which communicated

sequences from (or to) the surrounding environment. As a result, nothing of great import is actually being sacrificed in order to support implicit MAPS introduction.

Filters and Expressions Involving Multiple Sequences

In order to support the element at a time metaphor every sequence function (including those created by the *meta* sequence functions) is guaranteed to have the property of *registration*. As discussed above, a sequence is an ordered series of slots containing values. The registration property requires that the nth element in the sequence produced by a sequence function must be computed from the nth elements of the input sequences to that function. The computation can also involve state variables internal to the sequence function, but it cannot refer to any other elements in the inputs. The fact that the registration property is universally satisfied insures that it makes sense to talk about the interaction between the nth values in all of the sequences in a loop expression as *typical* because they are computed from each other.

The only sequence functions where there is any logical difficulty in satisfying the registration property are filters. It would be perfectly reasonable to say that a filter takes in a sequence and produces a shorter sequence containing only selected elements of the input sequence. From the point of view of the expressional metaphor there is nothing wrong with this definition, and there would be no difficulty in understanding a program like SUM-POSITIVE-EXPRESSIONAL based on this definition.

However, if filters produced shortened sequences, they would not satisfy the registration property. In order to satisfy this property, a filter is defined as producing a sequence which has exactly the same number of slots as its input with the selectivity of the filter encoded in the fact that some of the output slots are *empty*. The selected values remain in the same slots as in the input sequence. In order to make this work, loop expressions are defined as simply not operating on empty slots. This can be seen in the definitions of the various sequence functions and meta sequence functions presented above. The following general statement can be made: *A given loop subexpression is only executed on those cycles of the loop when values are available for all of the sequences it refers to.*

In order to appreciate the full impact of the definition of how filters operate, one must consider loop expressions involving several sequences. Consider the program LIST-EVEN-SQUARES. It takes in a list and returns a list. The output list contains an entry corresponding to each even number in the input. Each entry consists of the number followed by its square. For example when passed the argument (1 2 3 4) the program will produce the output ((2 4) (4 16)).

```
(defun list-even-squares (list)
  (letS ((integers (Elist list)))
    (Rlist (list (filterS #'evenp integers)
                 (* integers integers)))))
```

In the program, the function LIST is implicitly MAPSed over two sequences. The first is generated by enumerating the elements in the input and selecting the even elements (e.g., [_ 2 _ 4]). The second is generated by squaring all of the elements in the list (e.g., [1 4 9 16]). The registration between the two sequences is maintained by the fact that the missing elements in the filtered sequence are represented as empty slots. The function LIST is only executed when values are available in both sequences i.e., only for even elements of the list. The output of the implicit MAPS is a sequence which has values in it corresponding to the times when LIST was executed (e.g., [_ (2 4) _ (4 16)]). When RLIST reduces this sequence to a list it ignores these empty slots.

The registration property makes loop expressions easy to understand and compile; however, it is significantly restrictive. The key limitation is that there are a number of quite logical operations on sequences which cannot be supported. In particular, operations which disturb the ordering of the slots are prohibited.

For example, merging sequences, concatenating sequences, changing the order in a sequence (e.g., reversing it), etc. Such complex operations are not supported because the overhead associated with supporting them is not warranted by the rather infrequent need for them. When they are needed, other loop representations should be used.

Termination

As discussed above, termination presents problems from the point of view of the expressional metaphor. A loop expression is defined to terminate as soon as the shortest sequence in the expression is exhausted. This definition really only makes sense from the point of view of the element at a time metaphor. From the point of view of the latter metaphor, a loop expression is executed by computing all of the sequences in it in parallel an element at a time. The loop stops as soon as any sequence runs out of elements.

An interesting aspect of termination is the way it interacts with the registration property. Suppose, for example, that filters produced sequences of reduced length as their output. In this situation, a filter might well produce as the third and final element of its output the sixth element of its input. The definition of termination above would require that the loop stop after three cycles. However, this is a contradiction because the loop must run for at least six cycles in order to generate the sixth input to the filter so that the filter can produce its third output.

Note that all of the sequence functions and meta sequence functions (with the exception of `TRUNCATES`) are carefully restricted so that the lengths of their output sequences (if any) are exactly the same as the minimum of the lengths of their input sequences (if any). This is done so that truncators will be the only sequence functions which ever trigger termination. As a result, reasoning about termination can focus on the truncators in a loop.

At-start, At-end, & At-unwind

In addition to element at a time computation on sequences, many sequence functions specify initializing computation which is performed before the loop as a whole gets underway and/or epilog computation which occurs after the main body of the loop has terminated. Three additional meta sequence functions are provided which make it possible for users to specify computation to be performed at these times.

The meta sequence function (`AT-END` *function arg...*) specifies that *function* should be executed after the loop terminates. All of the *args* and any free variables referenced by *function* must hold unitary values. The value produced by applying *function* to *args* is returned as the unitary value of the form as a whole. As an example, consider the following definition of `RLIST`. A reducer is used to `CONS` together the items in the input sequence. After the reduction is completed, the function `NREVERSE` is used to reverse the order of the list so that the correct result is returned.

```
(defunS Rlist (&sequence items)
  (at-end #'nreverse (reduceS #'xcons nil items)))
```

The meta sequence function (`AT-UNWIND` *function arg...*) is just like `AT-END` except for two things. First, it produces no result at all. It is executed for side-effect only. Second, it will be executed no matter how the loop is terminated. The next section describes several situations in which unusual loop exits prevent `AT-END` computation from being performed. As an example of `AT-UNWIND`, consider the following definition of `RFILE`. A unitary auxiliary variable is used to hold the file object which is opened to receive the output. This file is closed when the output is completed. `AT-UNWIND` is used because it is important to close the file no matter how the loop is exited. Note that `AT-END` is used to specify that `T` should be returned as the result of `RFILE`.

```
(defunS Rfile (name &sequence items &aux &unitary (file (open name 'out)))
  (at-unwind #'(lambda () (close file)))
  (mapS #'(lambda (item)
            (let (prinlevel prinlength)
              (print item file)))
        items)
  (at-end #'(lambda () T)))
```

The meta sequence function (AT-START *function arg*...) is exactly the same as AT-END except that *function* is executed before the loop begins rather than after it terminates. The initialization of the auxiliary variable FILE in RFILE is an implicit example of AT-START computation. It could be made explicit as follows:

```
(defunS Rfile-explicit-at-start (name &sequence items &aux &unitary file)
  (at-start #'(lambda () (setq file (open name 'out))))
  (at-unwind #'(lambda () (close file)))
  (mapS #'(lambda (item)
            (let (prinlevel prinlength)
              (print item file)))
        items)
  (at-end #'(lambda () T)))
```

The examples above illustrate the most important use of AT-START, AT-END, and AT-UNWIND. They are used to include initializing and epilog computation as part of an individual sequence function. This extends the range of loop computation fragments which can be expressed as sequence functions. For example, RFILE would be conceptually much less useful if it did not encapsulate the actions of opening and closing the file into the same unit with printing out the objects.

The program PRINT-LIST-SUM illustrates the use of AT-START and AT-END inside of a LETS. The first line in the body of the LETS is executed only once at the start of the loop and prints a heading. The last line is executed only once at the end and prints the sum. The middle two lines are executed on every cycle of the loop and print out the integers in the list separated by spaces.

```
(defun print-list-sum (list)
  (letS ((x (Elist list)))
    (at-start #'(lambda () (format T "~%Integers: ")))
    (mapS #'(lambda (z) (format T "~D" z)) x)
    (mapS #'(lambda () (format T " ")))
    (at-end #'(lambda (z) (format T "~%Their Sum: ~D~%" z)) (Rsum x))))
```

The output produced by (print-list-sum '(1 2 3 4))

```
Integers: 1 2 3 4
Their Sum: 10
```

In order to make it easier to write loops like the one above, the rule for implicit MAPS introduction is extended by saying that *every* top level unitary expression in a LETS or DEFUNS body is MAPSed *if possible.* The only time it is not possible is if (like the last line of the LETS below) the expression refers to a value which is produced by a reducer and therefore not available until after the loop is completed. In this case, AT-END is implicitly introduced. Note that neither AT-START nor AT-UNWIND is ever implicitly introduced.

```
(defun print-list-sum-implicit (list)
  (format T "~%Integers: ")
  (letS ((x (Elist list)))
    (format T "~D" x)
    (format T " ")
    (format T "~%Their Sum: ~D~%" (Rsum x))))
```

Looking at the program PRINT-LIST-SUM-IMPLICIT above (which is exactly equivalent to PRINT-LIST-SUM) several points should be considered. The computation to be performed AT-START is simply placed outside and before the LETS. The second line in the LETS is implicitly MAPSed even though it refers to no sequence values at all. The third line is implicitly AT-ENDed because it refers to the output of a reducer. It should be noted that it is almost never necessary to actually write an explicit MAPS, AT-START, or AT-END.

Done

In addition to using the meta sequence function TRUNCATES to limit the length of a sequence, a loop can also be terminated by executing the special form DONE. Consider the program SUM-INITIAL. It takes in a list and adds up any initial group of numbers (e.g., (SUM-INITIAL '(1 2 A 4)) returns 3). The program works by enumerating the elements in the list and summing them up, but terminating the loop as soon as a non-number is encountered. The form (DONE) causes the immediately enclosing loop to terminate normally -- any AT-END loop computation which has been specified is performed, and the return value which is specified by the last line is returned (here the sum). Note that DONE only makes sense from the point of view of the element at a time metaphor, it does not fit into the expressional metaphor at all.

```
(defun sum-initial (list)
  (letS ((x (Elist list)))
    (cond ((not (numberp x)) (done)))
    (Rsum x)))
```

DONE can also be called with one or more arguments. In this case the loop is immediately terminated and the specified values are returned. When DONE is used in this way, it overrides the outputs specified in the last line of the LETS and any AT-END computations are not performed. An example of this use of DONE is shown in the program FIND-POSITIVE which returns the first positive number in a list.

```
(defun find-positive (list)
  (letS ((x (Elist list)))
    (cond ((plusp x) (done x)))))
```

The use of DONE is also illustrated by the following sequence functions. The sequence function ROR computes the OR of a sequence in the obvious way by successively ORing each value into a state variable. The first non-NIL value encountered is returned. The sequence function ROR-FAST also returns the first non-NIL value encountered; however, it causes the loop as a whole to terminate as soon as this value is found. Note the way the DONE overrides the NIL which is returned AT-END if no non-NIL items are found.

```
(defunS Ror (&sequence item)
  (reduceS #'or nil item))

(defunS Ror-fast (&sequence item)
  (cond (item (done item)))
  (at-end #'(lambda () nil)))
```

In general, ROR-FAST is more efficient than ROR; however, when you use it you must consider the effect that it will have on the rest of the loop it is used in. For example, because its operation is peremptorily terminated by the ROR-FAST, the program PRINT-LIST-OR-BUGGY neither prints out all of the elements in the list, nor prints out the summary line AT-END. In order to operate as intended, it needs to use ROR instead of ROR-FAST.

```
(defun print-list-or-buggy (list)
  (format T "~%Elements: ")
  (letS ((x (Elist list)))
    (format T "~A " x)
    (format T "~%Their Or: ~D~%" (Ror-fast x))))
```

The output produced by (print-list-or-buggy '(NIL A NIL B))

```
Elements: NIL A
```

It should be noted that you can cause the premature termination of a loop in other ways which are outside the scope of the LETS macro package. For example, you can wrap the loop in a PROG and then do a RETURN or GO from inside the loop to outside the loop. (Note that the loop expression itself is implemented by means of a PROG. In the LispMachine version (but not the MacLisp version), this PROG is named T in order to reduce interference with user specified RETURNs.) Similarly, you can do a THROW from inside the loop to some CATCH outside the loop. An important aspect of these kinds of exits is that they do not cause normal termination of the loop. No AT-END loop computation will be run, and the return value is directly specified by the RETURN or THROW.

Restart

It is occasionally convenient to be able to restart a loop at the beginning. The function RESTART reinitializes the immediately enclosing loop and causes it to start execution again from the beginning. The use of this function is illustrated in by the program RELAX.

```
(defun relax (graph function)
  (letS ()
    (cond ((Ror (funcall function (Egraph graph))) (restart)))))
```

The program RELAX takes in a graph and a function. The function is assumed to take in a node of the graph and perform some computation which may or may not result in side-effects which alter the node. If it alters the node then the function returns T, otherwise it returns NIL. RELAX repetitively applies the function to the nodes of the graph until the graph reaches a quiescent state where the nodes are no longer changing. A typical example of the way RELAX could be used would be to propagate some information through the graph.

The program works through multiple passes over the graph. It is assumed that the sequence function EGRAPH enumerates the nodes of the graph. In each pass, the function is applied to all of the nodes in the graph. The program computes the OR of the results of all of these function applications. If any of them is T then the loop is restarted in order to begin another pass over the graph.

LetS

Each LETS form delineates the exact extent of a loop. All of the sequence expressions in it (including any expressions specifying values for the sequence variables being bound) are combined together into a single loop which is separate from all other loops. The requirement that the output of a LETS be a unitary value results from the fact that the each LETS is compiled into a separate loop and therefore cannot be allowed to return a sequence into the surrounding environment.

LETS is defined to be a rigid boundary in order to better support the element at a time metaphor. Each LETS delineates a set of sequences which will be processed in parallel. This is important for clarifying concepts such as AT-START and AT-END. In addition, as will be discussed in detail below, it is even more important when considering side-effects (such as input/output) and nested loops.

It is typical for LETS to be used for loops which have a very strong element at a time flavor. In general, heavy use is made of implicit MAPS and AT-END in order to express these loops. The program INVENTORY-REPORT below shows a typical example of using LETS for this kind of loop. The program reads in a file of inventory records and prints out a report. Each record is a list of four fields: the name of the inventory item, the quantity on hand, the minimum acceptable quantity on hand, and the unit price. For each item the report prints out its name, how many are on hand, and the valuation of these items based on the specified price. The last line of the output reports the total valuation of all of the items. In addition to the above, the report prints out a notification in front of each item which is understocked indicating how many should be ordered.

Sample Inventory File Contents

```
("Widget"  8.    8.  20.5)
("Frob"    2.    9.  9.68)
("Thingy"  312. 40. 19.65)
("Dingus"  0.   20.  5.25)
("Whatsit" 3.    7.  5.67)
```

Resulting Printout

```
          Inventory Report

Order?      Name      On Hand  Valuation
            Widget        8     $164.00
Order:    7 Frob          2      $19.36
            Thingy      312    $6130.80
Order:   20 Dingus        0       $0.00
Order:    4 Whatsit       3      $17.01

          Total Valuation:   $6331.17
```

Looking at the loop in the program, note the use of destructuring and sequential assignment in the bound variable value pairs. In the first line of the LETS, the sequence variable NAME is bound to a sequence of the first field of each record, the variable QUANTITY is bound to a sequence of the second field of each record, etc. The variable VALUATION is bound to a sequence of products of PRICE and QUANTITY.

```
(defun inventory-report ()
  (with-open-file (report "inventory.report" ':out)
    (format report "~10T Inventory Report~2%")
    (format report "Order?     Name     On Hand  Valuation~%")
    (letS (((name quantity minimum price) (Efile "inventory.data"))
           (valuation (*$ price (float quantity))))
      (cond ((>= quantity minimum) (format report "~10X"))
            (T (format report "Order: ~3D" (- minimum quantity))))
      (format report "~X~10A~4D~2X~10<$~$~>~%" name quantity valuation)
      (format report "~%~11XTotal Valuation:~10<$~$~>" (Rsum$ valuation)))))
```

The body of the LETS prints the main part of the actual report. The first form prints the ordering notifications. It compares the quantity in stock with the minimum required and prints out the number to be ordered if the quantity is less than the minimum. The second form prints the main information about each inventory item. (Note that the FORMAT function is a Lisp function for creating formatted output. Like the Fortran construct it is modeled after, it is inscrutable but convenient.) Both of the first two forms in the body are implicitly MAPSed. The third form prints out the summary line at the end of the report. It is only executed once at the end of the loop because it uses the unitary output of the reducer RSUM$ (floating point sum).

An important thing to note about INVENTORY-REPORT is that although the process of actually printing out the report (i.e., opening the file, printing some initial lines, printing a group of internal lines, printing a final line, and then closing the file) is clearly a logically identifiable loop fragment, it is not represented as a sequence function. The problem is that, unlike the simpler actions represented by RFILE, there are so many ways in which the items to be printed, and the format for printing them, can vary that there is very little constant structure which could be captured in a sequence function. Basically, the only thing which is common between different instances of this fragment is opening and closing the file which is already captured in the form WITH-OPEN-FILE.

A key aspect of LETS is that even though the operation of actually printing the report are not represented as a sequence function, LETS makes it possible for them to be conveniently expressed. This is done in basically the same way that it would be done in an ordinary looping notation i.e., by distributing the parts of the computation into places where they will be executed in the correct situations. It must be said that this makes this particular fragment no easier to understand than it would be in an ordinary looping notation. However, the loop as a whole is more understandable because much of the computation is represented concisely in terms of sequence functions. The ability to mix computations which are not specified as sequence functions into a loop expression is another important capability which is facilitated by the element at a time metaphor. The issue of the kinds of loop fragments which cannot be represented will be discussed more fully in the section on the domain of applicability of the expressional loop notation.

Side-Effects

The behavior of side-effect producing operations (such as input/output) in a loop expression can only be understood from the point of view of the element at a time metaphor. This is another important reason why this metaphor is made a prominent part of the description of the notation.

The compilation process is constrained so that the order of execution in the loop produced is rigidly linked to the lexical order of expressions in the original loop expression. As a result, it is relatively easy to predict the consequences of side-effects as long as you bear in mind the fact that processing is occurring an element at a time so that the side-effect operations are interleaved and that each one is executed many times. Consider the program INVENTORY-REPORT above. The two main output statements are each executed once on each cycle

of the loop. The final output statement is executed only once after the loops terminates.

Note that the requirement that every unitary expression in a LETS be MAPSed if possible was introduced in order to make side-effects easier to understand. One might have said that an expression which neither uses nor produces sequence values should not be MAPSed since its value cannot change on different cycles of the loop. However, this would be missing the fact that if it has a side-effect (such as output to a file) this effect is probably desired on every cycle of the loop. A programmer can use AT-START in order to specify that something should only be executed once.

Most side-effects interact with the expressional notation in straightforward ways and can easily be understood as outlined above. However, there are some situations where things are not so clear.

Side-Effects and Termination

In order to understand how side-effects interact with termination, one has to be aware of exactly when termination will occur. For example, consider the program PRINT-LIST below. This function prints all of the items in a list preceding each one with an index of its position in the list.

```
(defun print-list (list)
  (letS ((i (generateS #'1+ 1))
         (x (Elist list)))
    (format T "~%Item ~D:" i)
    (format T " ~A" x)))
```

The output produced by (print-list '(A B C)):

```
Item 1: A
Item 2: B
Item 3: C
```

There is one potential pitfall which the user must be aware of. A loop is terminated *immediately* upon discovering that one of the sequences has been exhausted. As a result of this, unless the termination test happened to be the first thing executed on that cycle of the loop, some things will get executed on that last cycle, and others will not. In particular, all and only those expressions which lexically precede the termination will be executed. For example, consider the program PRINT-LIST-BUGGY. (Note that although no sequence variables are bound, a LETS is required in this program in order to specify that the two FORMATs should be executed in a single loop instead of in two separate loops. The LETS also specifies that the FORMATs should be MAPSed.)

```
(defun print-list-buggy (list)
  (letS ()
    (format T "~%Item ~D:" (generateS #'1+ 1))
    (format T " ~A" (Elist list))))
```

The output produced by (print-list-buggy '(A B C)):

```
Item 1: A
Item 2: B
Item 3: C
Item 4:
```

This program does not produce the same output as PRINT-LIST. The problem is that it does not discover that the list has been exhausted until after the first FORMAT has been executed on the last cycle of the loop. Note that this problem cannot be avoided by any straightforward change to the definition of LETS. You could not say that nothing in a cycle will be executed if any termination is triggered because some of the computation may be necessary in order to compute when to terminate. On the other hand, you could not say that everything will be executed on the cycle where termination occurs because typically some (or all) of the

iputation after the termination test will be in error if the test is true.

The programmer is capable of exercising control over this problem because, in the loop code which is duced, everything is evaluated in the order in which it appears in the original loop expression. As a result, ; always possible for him to get the termination tests to occur at the places he wants by correctly ordering forms in the LETS. For example, the ELIST is merely placed before the first FORMAT in PRINT-LIST. As sult, this is not really a severe problem; however, it is one to which the user must be sensitized.

On a deeper level, the real problem with PRINT-LIST-BUGGY is that neither it (nor for that matter PRINT-T) makes the logical relationship between the two FORMATs explicit. The correct thing to do is to group m together into a single form as in the function PRINT-LIST-BEST.

```
(defun print-list-best (list)
  (letS ()
    (format T "~%Item ~D: ~A" (generateS #'1+ 1) (Elist list))))
```

Side-Effects Between Sequence Functions

As mentioned above, the expressional notation attempts to maintain the property of decomposability of p expressions whenever possible. An important feature of this is that any internal state variables of a uence function are hidden from view and cannot be modified by SETQs, or the like, in a loop expression. fortunately, side-effect producing functions such as RPLACD are capable of modifying the values of state iables without having to actually refer to the variables themselves. If such side-effect functions are being d, then the programmer must take care that this kind of problem does not arise.

The problem is illustrated by the program DASH-LIST-BUGGY. The purpose of this program is to take in a (e.g., (A B C)) and put a dash after each entry in it (e.g., to produce (A - B - C -)). It attempts to do by side-effect as follows. It enumerates each of the sublists in the original list (e.g., [(A B C) (B C) (C)]) splices in a dash after the first element of each sublist (e.g., producing [(A - B C) (B - C) (C -)]).

```
(defun dash-list-buggy (list)
  (letS ((sublist (Esublists list)))
    (rplacd sublist (cons '- (cdr sublist))))
  list)
```

Particularly from the point of view of the expressional metaphor, the above algorithm sounds very isible; however, it doesn't work. What actually happens is that the program goes into an infinite loop ;ing in dashes after the first item in the input list. For example, if the loop starts with the list (A B C) the first sublist is (A B C). The RPLACD alters this sublist to (A - B C) and therefore the list itself to - B C). So far this is all as intended. Unfortunately, an internal variable in ESUBLISTS has a pointer the list in order to keep track of what sublist to enumerate. The list is altered *before* the second sublist is ally enumerated and as a result (- B C) gets enumerated as the second sublist instead of (B C).

t is possible to construct a loop expression for this algorithm which will work more or less as intended. example, the program DASH-LIST1 combines everything into one enumerator which enumerates the next ist before the RPLACD operation. Alternatively, DASH-LIST2 uses a modified enumerator which makes vances for the actions of the RPLACD.

```
(defun dash-list1 (list)
  (letS ()
    (enumerateS #'null
                #'(lambda (l) (prog1 (cdr l) (rplacd l (cons '- (cdr l)))))
                list))
  list)

(defun dash-list2 (list)
  (letS ((sublist (enumerateS #'null #'cddr list)))
    (rplacd sublist (cons '- (cdr sublist))))
  list)
```

However, due to the antagonistic interaction between the RPLACD and the enumerator, there is no aesthetic way to express the stated algorithm using the expressional notation. As will be discussed in more detail below, this is one of the kinds of algorithms for which the expressional notation is not intended to be used.

Conversions and Coercions

Two sequence functions are available for converting between unitary values and sequences: GSEQUENCE which converts an object into an infinite sequence of that object, and RLAST which converts a sequence into a unitary object by taking its last element. It should be noted that the meta sequence function MAPS is like GSEQUENCE in many ways. If passed a unitary object it will also create an infinite sequence of that object. However, if you nest a unitary expression in MAPS it will be executed many times, while if you apply GSEQUENCE to the expression it will be evaluated only once. For example, VECTOR-NCONS initializes a vector by filling all of its slots with the same CONS cell. In contrast, VECTOR-NCONSES fills each slot with a different CONS cell.

```
(defun vector-ncons (vector)
  (Rvector vector (Gsequence (ncons nil))))

(defun vector-nconses (vector)
  (Rvector vector (mapS #'(lambda () (ncons nil)))))
```

In order to make things more convenient for the user, automatic type coercions are applied between sequences and unitary values. The most important coercion has already been discussed. Whenever a unitary expression is placed where a sequence value is required, MAPS is automatically introduced in order to convert it into a sequence expression. Note that GSEQUENCE is never automatically introduced and therefore VECTOR-NCONSES-IMPLICIT is equivalent to VECTOR-NCONSES, and not to VECTOR-NCONS.

```
(defun vector-nconses-implicit (vector)
  (Rvector vector (ncons nil)))
```

The places where sequence values are required are the sequence arguments to sequence functions and the expressions to be bound to values in a LETS. These coercions are illustrated by the following pairs of equivalent loop expressions.

```
        (Rlist l)
same as: (Rlist (mapS #'(lambda () l)))

        (letS ((x 1))
          ...)
same as: (letS ((x (mapS #'(lambda () 1))))
          ...)
```

In the reverse direction, whenever a sequence expression is placed where a unitary value is required, RLAST is automatically introduced to convert it into a unitary value producing expression. The places where

unitary values are required are the last expression in the body of a LETS and the value of loop expressions which appear in isolation in ordinary Lisp code. Examples are shown below.

```
          (letS ((x (Elist list)))
            (mapS #'print x))
same as:  (letS ((x (Elist list)))
            (Rlast (mapS #'print x)))

          (mapS #'print (Elist x))
same as:  (Rlast (mapS #'print (Elist x)))
```

Some of the other features of the expressional notation could also be looked at as coercions, for example, the automatic introduction of MAPS and AT-END around lines of a LETS. Taken together, these coercions have no semantic import -- they do not make it possible to express anything which could not be expressed without them. However, they do make it significantly more convenient to specify many kinds of loops.

Nested Loops

Like any looping notation, the expressional notation can be used to express nested loops. Consider the program SUM-LISTS-IN-LIST1. It takes in a list of lists of integers (e.g., ((1 2) (3 4))) and returns a list of the sums of these lists (e.g., (3 7)). The outer loop enumerates the lists of numbers in the list supplied as the input to the function as a whole. The inner loop adds up the numbers in these sublists. The outer loop then CONSes these numbers up into a list to be returned.

```
(defun sum-lists-in-list1 (list-of-lists)
  (letS ((entry (Elist list-of-lists)))
    (setq entry (mapS #'(lambda (l) (Rsum (Elist l))) entry))
    (Rlist entry)))
```

In the program, MAPS is used to apply the inner loop to each list of numbers in turn. The LAMBDA used with the MAPS delineates the boundary of the inner loop. This could also be done by wrapping a LETS around the inner loop (which would then be implicitly MAPSed) as in SUM-LISTS-IN-LIST2.

```
(defun sum-lists-in-list2 (list-of-lists)
  (letS ((entry (Elist list-of-lists)))
    (setq entry (letS () (Rsum (Elist entry))))
    (Rlist entry)))
```

Though relatively clear, both of the above programs are somewhat cumbersome in appearance. If the (RSUM (ELIST ...)) were in isolation, there would be no need to wrap it in either a MAPS or LETS. The same is true here. The algorithm can be more conveniently expressed as shown in SUM-LISTS-IN-LIST3.

```
(defun sum-lists-in-list3 (list-of-lists)
  (letS ((entry (Elist list-of-lists)))
    (setq entry (Rsum (Elist entry)))
    (Rlist entry)))
```

Note that from the point of view of the element at a time metaphor, the body of the LETS is describing what happens to a typical value of ENTRY. This typical value is unitary and therefore it makes perfect sense to say that (RSUM (ELIST ...)) is applied to it. However, as written, the loop expression contains type conflicts. The sequence ENTRY is supplied where ELIST expects a unitary input, and the unitary output of RSUM is assigned to the sequence ENTRY. In order to deal with this, an automatic conversion is applied which parses each loop expression looking for matched pairs of type conflicts like these. These conflicts are then resolved by separating out a nested loop which is MAPSed over the input sequence.

Unfortunately, there are several major problems involved with the automatic introduction of nested loops

as described above. The first is that although the third version of the program above is arguably more readable than the first two versions, the process of representing a loop more and more compactly can easily by carried to excess. For example, the next two versions of the program are more compact and specify exactly the same computation. However, it is questionable whether they are easier to understand. Going beyond this, more complex programs (such as triplely nested loops) become virtually incomprehensible if rendered in such a dense expressional style.

```
(defun sum-lists-in-list4 (list-of-lists)
  (letS ((entry (Elist list-of-lists)))
    (Rlist (Rsum (Elist entry)))))

(defun sum-lists-in-list5 (list-of-lists)
  (Rlist (Rsum (Elist (Elist list-of-lists)))))
```

Another problem with the automatic creation of nested loops is that although the required parsing is trivial in simple cases like the above, it is unfortunately quite complex in the general case. One reason for this is that there is considerable interaction with the type coercion processes described above. Another stems from the fact that parsing also has to deal with the related phenomenon illustrated in the program SUM-COPY-OF-LIST below. This program copies a list of integers and then computes the sum of the integers. Note that there are no type conflicts in this program, and that the initial copying of the list is not a nested loop. It is executed in its entirety before the summation loop begins. Nevertheless, the copying loop has to be located and separated from the rest of the loop since it is computed separately. The need to do this further complicates the parsing process. All in all the parsing/coercion process ends up being by far the most complex part of the compilation process.

```
(defun sum-copy-of-list (list)
  (Rsum (Elist (Rlist (Elist list)))))
```

A much bigger problem with the automatic introduction of nested loops is that the complex interactions with other coercions are not just hidden inside the compilation process -- they can lead to considerable confusion regarding seemingly simple loop expressions. For example, consider the program ZERO-MATRIX. As rendered below it has a simple explicit nested loop which sets all of the elements of an array to zero. If one tries to express this more compactly, things rapidly become complicated.

```
(defun zero-matrix (A)
  (letS ((i (Erange 0 (1- (array-dimension-n 1 A)))))
    (letS ((j (Erange 0 (1- (array-dimension-n 2 A)))))
      (aset 0 A i j))))
```

The variable J is only used once in the inner loop, so the program ZERO-MATRIX-NO-J is equivalent to ZERO-MATRIX. However, if you omit the inner LETS as in ZERO-MATRIX-BUGGY you no longer have an equivalent program. This program is is not equivalent to ZERO-MATRIX, but rather to ZERO-DIAGONAL.

```
(defun zero-matrix-no-j (A)
  (letS ((i (Erange 0 (1- (array-dimension-n 1 A)))))
    (letS ()
      (aset 0 A i (Erange 0 (1- (array-dimension-n 2 A)))))))

(defun zero-matrix-buggy (A)
  (letS ((i (Erange 0 (1- (array-dimension-n 1 A)))))
    (aset 0 A i (Erange 0 (1- (array-dimension-n 2 A))))))

(defun zero-diagonal (A)
  (letS ()
    (aset 0 A (Erange 0 (1- (array-dimension-n 1 A)))
              (Erange 0 (1- (array-dimension-n 2 A))))))
```

The problem is that there is a considerable amount of coercion going on in ZERO-MATRIX-NO-J which is no longer forced in ZERO-MATRIX-BUGGY. In particular: the ASET is MAPSed over the sequence created by the inner ERANGE; in conjunction with this MAPS, each individual value of I is in effect converted into a sequence of identical values; and RLAST is used to convert the sequence of values created by the MAPSed ASET into a unitary return value. In ZERO-MATRIX-BUGGY everything can be interpreted much more simply by assuming that the two ERANGEs are being executed in parallel. It is a general feature of LETS that it only creates nested loops when it is absolutely necessary -- it always tries to combine everything in its body into a single loop.

Given the difficulties it causes, the obvious question is why support the automatic introduction of nested loops? The problem is that the simple cases of implicitly nested loops are so logically compelling that they cannot be ignored. It seems to be an obvious benefit to be able to use a simple loop expression (such as (RSUM (ELIST X))) in isolation in a program. Add to this the fact that the element at a time metaphor suggests thinking about the interior of a loop as a specification for what happens to typical (unitary) values of the sequences in it, and programs like SUM-LISTS-IN-LIST3 seem too reasonable to prohibit.

A Large Example

To conclude the description of the features of the expressional loop notation, this section presents a larger example. The example is a data abstraction which implements sets of symbols as bit vectors. The abstraction not only makes available some ordinary functions for operating on these sets, but some sequence functions as well.

Sets are represented as bits packed into a single integer. The size of the sets is limited by the number of bits in an integer (e.g., 24 bits on the LispMachine). The global variable *BSET-DOMAIN* stores the correspondence between potential set elements and bit positions. This mapping is represented by a vector of CONSes. The index of a CONS in the vector indicates the bit position which is being described. The CAR of the CONS holds the symbol which corresponds to the bit position. The CDR of the CONS holds the representation for a set which has only that one symbol in it (i.e., an integer with only the one corresponding bit on). The variable *BSET-DOMAIN* is initialized to a vector of conses of NIL and the appropriate single element sets. Note that the unit sets are created by a special generator which starts with an integer with a 1 in bit position 0 and then rotates this bit around from position to position.

```
(defvar *bset-domain*
  (Rvector (array nil T 24) (cons nil (generateS #'(lambda (x) (rot x 1)) 1)))
  "The bset domain element mapping.")
```

The global variable *BSET-INDEX* keeps track of the largest bit position used so far. The number -1 is used to represent the fact that no bit positions have been used yet.

```
(defvar *bset-index* -1 "The largest bit position used so far.")
```

The function BSET-RESET is used to reinitialize these variables. It MAPSes over the vector in *BSET-DOMAIN* setting the CAR of each CONS cell to NIL, and sets *BSET-INDEX* to -1.

```
(defun bset-reset ()
  (letS ((item (Evector *bset-domain*)))
    (setf (car item) nil))
  (setq *bset-index* -1))
```

The function BSET-UNITSET takes in a symbol and returns the unit set corresponding to it. It issues an error if the symbol is not representable as a unit set (i.e., if it is not in the vector *BSET-DOMAIN*). It uses ROR-FAST (which as described above, computes the OR of the items in a sequence, stopping as soon as a non-NIL item is encountered) in order to look for the symbol in *BSET-DOMAIN* returning the corresponding unit set as soon as it is found. Note that the COND in the ROR-FAST is implicitly MAPSed.

```
(defun bset-unitset (symbol)
  (or (letS ((item (Evector *bset-domain* 0 *bset-index*)))
        (Ror-fast (cond ((eq (car item) symbol) (cdr item)))))
      (error "symbol not in bset domain" symbol)))
```

The function BSET-ADD-DOMAIN-ELEMENT takes a symbol and enters it in *BSET-DOMAIN* so that it can be used in the bit vector sets. If the symbol is not already in the domain, and if there is an available bit position, then the program increments *BSET-INDEX* and stores the symbol in the appropriate CONS cell in *BSET-DOMAIN*.

```
(defun bset-add-domain-element (symbol)
  (cond ((Ror-fast (eq symbol (car (Evector *bset-domain* 0 *bset-index*))))
         ((> *bset-index* 22) (error "bset domain size exceeded" nil))
         (T (incf *bset-index*)
            (setf (car (aref *bset-domain* *bset-index*)) symbol))))
```

As examples of the kind of ordinary functions which would be implemented as part of the data abstraction consider the following four. The first three are examples of the operations for which the bit vector implementation is particularly efficient. Intersection, union, and the test for equality between two sets can all be implemented as single operations independent of how many symbols are in the sets operated on.

```
(defun bset-intersect (bset1 bset2)
  (logand bset1 bset2))

(defun bset-union (bset1 bset2)
  (logior bset1 bset2))

(defun bset-equal (bset1 bset2)
  (= bset1 bset2))

(defun bset-mem (symbol bset)
  (not (zerop (bset-intersect (bset-unitset symbol) bset))))
```

The next four definitions are examples of the kind of sequence functions which would be provided as part of the data abstraction. The first two implement reducers which can be used to take the intersection and union of sequences of bit vector sets. The third (EBSET) takes in a bit vector set and creates a sequence of the symbols in that set. The last (RBSET) performs the inverse operation, taking in a sequence of symbols and creating a set by taking the union of the corresponding unit sets.

```
(defunS Rbset-intersect (&sequence bset)
  (reduceS #'(lambda (x) (bset-intersect x bset)) -1))

(defunS Rbset-union (&sequence bset)
  (reduceS #'(lambda (x) (bset-union x bset)) 0))

(defunS Ebset (bset)
  (car (filterS #'(lambda (x) (not (zerop (bset-intersect (cdr x) bset))))
                (Evector *bset-domain* 0 *bset-index*))))

(defunS Rbset (&sequence symbol)
  (Rbset-union (bset-unitset symbol)))
```

The example above is a particularly good one in that it shows the expressional notation being used to represent a variety of loops which are small and simple. This is the application for which the notation has been specifically designed.

III - Evaluating the Expressional Notation

One way to summarize the expressional loop notation is as a collection of basic ideas. First, there are three themes which underlie the notation.

The Expressional Metaphor - The idea that loops can be expressed as compositions of fragments of looping behavior is the fundamental motivation behind the notation.

The Element at a Time Metaphor - The additional metaphor that a loop can be conveniently specified as a set of operations on typical elements also underlies the notation as a whole.

Efficient Compilation - From the beginning, it was decided that it had to be possible to compile the notation into efficient looping code. This effected many of the design decisions.

There are six basic features of the notation which together support these themes.

Sequence Functions - These embody the fundamental notion of a fragment of looping behavior. The fact that they look and can be reasoned about essentially just like ordinary functions supports the expressional metaphor. Restrictions on the kinds of sequence functions allowed (e.g., the requirement for registration between elements of their inputs and outputs) support the element at a time metaphor and efficient compilation.

Sequences - These are the mode of communication between sequence functions. The fact that they look like and can be reasoned about much of the time just like ordinary aggregate data objects supports the expressional metaphor. The fact that they are defined to be one dimensional series of slots containing unitary values where each slot corresponds to one cycle of the loop which will eventually be produced is the fundamental underpinning for the element at a time metaphor and is essential for efficient compilation.

User Definition of Sequence Functions - The fact that the user can define his own sequence functions in analogy with the definition of ordinary functions greatly extends the utility of the notation.

Meta Sequence Functions - These make it possible to specify new kinds of operations on sequences. On the one hand, they provide a very convenient mechanism for this specification. One the other hand, they embody the restrictions which are necessary in order to insure that it will be possible to efficiently compile the specified operations. In this context it is important that the notation does not provide any more general method for specifying sequence computations.

Loop Expression Blocks - Calls on LETS serve two basic purposes: delineating groups of loop expressions which are to be combined into a single loop, and supporting the notion of variables which have sequences as their values. The body of such a block is the place where the element at a time metaphor is most prominent.

Coercions - The existence of coercions such as the automatic introduction of MAPS is an important underpinning for the element at a time metaphor. Other coercions such as the detection of nested loops and automatic conversions between sequences and unitary values exist merely as a convenience for the user. Note that in order to make the above coercions practical, variables containing sequences have to be readily identifiable as such.

In order to investigate the efficacy of the expressional loop notation as a whole, one must look at it from several points of view. In particular, one must evaluate the kind of loops it can be applied to, how efficiently it can be executed, and how easily it could be applied to other languages besides Lisp.

Domain of Applicability

The expressional loop notation is oriented towards the kinds of straightforward loops which are most common. In order to make it easier to express these loops, it deliberately sacrifices more general applicability. As a result, there are a number of situations where the expressional notation is not appropriate.

The basic approach of the notation is to express a loop as a composition of fragments of looping behavior represented as sequence functions. There are two main situations in which this approach is ineffective: when a loop cannot be separated into multiple fragments, and when the notion of a sequence function is not capable of expressing the required fragments.

It is quite possible that even a large loop will not be decomposable into fragments. In order to break a loop down into two fragments A and B, it must be the case that A and B are both self contained units. In particular this means that there can be no interaction between A and B other than data flow from A to B. Note that there cannot be any data flow from B to A. In some loops, all of the computation is linked together in a tight net of data flow. In that case it cannot be decomposed. For example, consider the program BINARY-MEM which tests whether a given integer is in a sorted vector of integers by doing a binary search.

```
(defun binary-mem (integer vector)
  (prog (left mid right item)
        (setq left 0)
        (setq right (1- (array-length vector)))
      L (cond ((> left right) (return nil)))
        (setq mid (// (+ left right) 2))
        (setq item (aref vector mid))
        (cond ((> item integer) (setq right (1- mid)))
              ((< item integer) (setq left (1+ mid)))
              (T (return T)))
        (go L)))
```

The program cannot be decomposed into a composition of fragments because each part affects every other part. The values of LEFT and RIGHT are used to compute MID which is used to compute ITEM which is used in a test which determines the next values of LEFT and RIGHT. Because it cannot be decomposed, there is no way to write the program any more clearly using the expressional notation. The best that could be done would be to write the program as one huge sequence function.

Often side-effects tie together parts of a loop which might appear to be separable. An example of this was shown in the section on side-effects above. The basic algorithm presented there for adding dashes into a list cannot be decomposed because the RPLACDs performed on the sublists which are enumerated modifies the state of the enumerator. The only solutions are either to represent the program as a single fragment as in DASH-LIST1, or to write a program like DASH-LIST2 where the loop appears to have been decomposed but actually has not. Both approaches are unsatisfactory. Neither program is particularly easy to understand, and the second program violates the basic spirit of the expressional notation. It would be better to refrain from using the expressional notation for this kind of program.

The expressional loop notation is also limited in the kind of loop fragments which it can represent. As described above, one area of limitation is a result of the simple notion of sequence which underlies the notation. This makes it impossible to express fragments which alter the order of elements in a sequence or which merge sequences.

The only facilities available for creating fragments are the meta sequence functions. Experience has shown that these are capable of creating a wide range of useful fragments. However, there are a variety of plausible fragments which cannot be created. For example, ENUMERATES always creates fragments where the termination test is performed at the start of each cycle of the loop. It is not possible to create a fragment where the termination test is performed at the end of each cycle.

Another example of a fragment that cannot be represented as a sequence function is the idea of doing output to a report file as discussed above in conjunction with the program INVENTORY-REPORT. An important thing to notice from that example however, is that LETS makes it possible to combine a fragment like this one with a loop in expressional notation. This considerably extends the domain of applicability of the notation. A vital feature of this is that the notation acts to protect the semantic integrity of the standard fragments when a non-standard fragment is added. The primary way it does this is by hiding the internal state variables of the standard fragments, so that non-standard fragments cannot modify them.

Another more fundamental reason why the expressional loop notation may not be appropriate is that some other paradigm may be more appropriate. For example, consider the function GCD. Writing it as a recursive program makes it very easy to understand because the structure of the program exactly mirrors the structure of the standard proof of correctness for the algorithm. No iterative rendition would be as clear.

```
(defun gcd (x y)
  (cond ((< x y) (psetq x y y x)))
  (let ((r (remainder x y)))
    (cond ((zerop r) y)
          (T (gcd y r)))))
```

In addition, it should be noted that unlike some looping notations the expressional notation does not handle anything but simple loops. For example, it does not support multiple entry points nor, by itself, exits to multiple points.

Efficient Execution

There are two principal ways in which the expressional notation could be executed: *direct execution*, and *conversion to iterative loops*. The most straightforward way would be to just implement sequences as normal data objects and the sequence functions as normal functions. Loop expressions could then be evaluated just like any other expressions. This direct execution approach is taken by API. [10]. On the other hand, a compilation process can be used to convert loop expressions into ordinary iterative loops which operate in an element at a time fashion. This conversion approach is used by LETS and the languages Hibol [12,13] and Model [11].

The main advantage of direct execution is that it is easy to implement. In particular, it is very easy to see how it directly supports the expressional metaphor. The main disadvantage of direct execution is that, in comparison with ordinary iterative loops, it imposes very large time and space overheads.

The main advantage of the conversion approach is that it is capable of creating very efficient code. In fact there is no reason in principle why there has to be any time or space overhead at all. There are, however, two drawbacks to this approach. First, the conversion process can be quite complex. Just how complex depends on exactly what facilities are supported by the notation. Second, the conversion process is fundamentally related to the element at a time viewpoint. As we have seen, due to issues like termination and side-effects, this viewpoint cannot be hidden from the user. As a result, the user has to keep this metaphor in mind as well as the simple expressional metaphor.

When designing the expressional notation it was felt that the issue of efficiency could not be ignored. As a result, the notation was designed from the beginning with conversion in mind. This had two major effects on the design. First, whenever a potential feature of the notation would have unduly complicated the conversion process it was discarded. Second, the element at a time metaphor was introduced as an explicit part of the motivation behind the notation. These precepts resulted in a notation which is in fact relatively straightforward to compile into efficient code.

The LETS macro package implements the notation using a straightforward set of macros which convert

each expression into an ordinary loop. No explicit representation for a sequence is ever required. The compilation process is described in detail in Appendix B. The next few paragraphs outline the major features of the process.

Each sequence function is represented by a data structure specifying some initialization computation to perform before the loop begins, some inside computation to perform repetitively on each cycle of the loop, and some epilog computation to perform after the loop terminates.

A composition of two sequence functions "`(A (B ...))`" is compiled by combining their parts together into a new compound sequence function. The resulting initialization, insides, and epilog are derived by concatenating the corresponding parts of `B` and `A`. The data flow from `B` to `A` is implemented by data flow from the inside part of `B` to the inside part of `A` in the new compound inside part.

When a loop expression is encountered, it is first parsed in order to locate all of the sequence functions in it. The meta sequence functions are implemented as macros which take their functional arguments and create an appropriate sequence function. As part of the parsing process, implicit `MAPS` introduction and other coercions are introduced. The loop expression is then compiled by combining all of the sequence functions in it together. Once this has been done, the resulting fragment is converted into an actual loop with the indicated parts.

Returning to a discussion of alternate implementation strategies, it should be noted that in order for direct execution to be used with the expressional notation the notation would have to be altered in several non-trivial ways. To start with, the notation supports potentially infinite sequences. For example, inside the typical enumerator is a generator creating an infinite sequence, and a truncator cutting this down to finite length. You cannot just compute the entire generated sequence before truncating it. The easiest way to deal with this is to follow the lead of APL and simply outlaw generators and infinite sequences, allowing only enumerators of finite sequences. However, as we have seen, subsidiary generators can be very convenient in programs such as `DIGITS-TO-NUMBER`.

A much more fundamental set of problems arises from the fact that the element at a time metaphor is fundamentally incompatible with direct execution. The behavior of termination and side-effects is completely different in the context of direct execution. In many situations it is not clear whether these alternate behaviors would be more or less useful. However, with regard to programs like `INVENTORY-REPORT` which are particularly well suited to the element at a time metaphor, they are quite likely to be less convenient. In any case, it is a fundamental change and much of the notation would have to be redesigned.

There has been a lot of interesting work which tries to chart a middle course between the simplicity of the direct execution approach and the efficiency of the conversion approach. One way in which this has been done is by representing sequences explicitly, but without trying to compute the elements in them until they are actually needed. This can be done explicitly through coroutines [7], or implicitly through lazy evaluation [4,6]. In LispMachine Lisp, this could be done by using streams to represent sequences. Note that this delayed evaluation approach is capable of dealing with infinite sequences as well as finite ones. Also note that from the point of view of what gets executed when, this approach entails a more or less complete conversion to element at a time processing.

Although delayed evaluation is more efficient (particularly in space) than direct execution in many situations it is still much less efficient than complete conversion. Several researchers have pursued an interesting mixed mode approach which provides an interpreted implementation where sequences are represented explicitly and, in addition, provides a compiler which performs conversions to eliminate intermediate sequences whenever possible.

The premier example of this has been the work on compilers for APL [2,5]. Optimizing APL compilers attempt to locate array expressions where the arrays are being used merely as intermediate sequences, and

then eliminate the actual computation of these arrays. When an expression corresponding to the kind of simple loop representable by the expressional notation is located, then it is easy to eliminate the intermediate arrays. Wadler's Listless Transformer [15] pursues a similar approach for compiling a Lisp-like language. It takes programs where finite intermediate sequences are represented as lists, and converts them into programs where these intermediate lists do not actually have to be created. The resulting programs can then be efficiently compiled by normal means.

Unfortunately, there are several inherent problems with the partial conversion approach. First, since direct execution of unconverted loop expressions must be supported, the notation must have all of the restrictions outlined above. Second, the only reason to pursue partial conversion is that the notation supports features which cannot be practically converted. Unfortunately this raises a whole new problem -- that of identifying what parts of what loops can be converted. In addition, steps have to be taken to interface loop expressions which have been converted with those which have not.

A third and much more serious problem is that in the presence of side-effects, conversion is not a correctness preserving process. The reason for this is that it entails a radical change in execution order from computing each sequence as a unit to processing several sequences an element at a time. To deal with this you either have to refrain from converting any loop containing side-effects (including input/output) or you have to specify that such loops will always be converted and require the user to think in terms of the element at a time metaphor. Note that the latter approach cannot be taken if it is possible for the user to write a side-effect containing loop which cannot be converted.

The approach taken here has been to avoid this kind of problem by simplifying the notation to the point where complete conversion is practical. The languages Hibol and Model support somewhat similar notations (described in greater detail below) which are also suitable for complete conversion.

Language Independence

All of the discussion above was couched in terms of Lisp, and the initial implementation of the expressional notation has been done for Lisp. However, none of the ideas presented here are inherently dependent on any specific language. It is true that it is particularly easy to make this kind of extension to the language Lisp. However, by modifying its compiler this could be introduced as an extension to any language. For example, you could add the expressional loop notation to the language Ada [1] by supporting the six basic features of the notation as follows:

Sequence Functions - As in the Lisp implementation, calls on sequence functions would look syntactically exactly like calls on other functions; however, they would be handled like macros in order to create loops as described above. A set of basic sequence functions would be provided as part of the standard environment.

Sequences - A new data type SEQUENCE OF would be added. This could be used to specify the data types of variables and of the arguments to sequence functions.

User Definition of Sequence Functions - A new kind of function declaration `SEQUENCE FUNCTION` would be added. Using this, sequence functions would be defined exactly like ordinary functions. These would be the only functions allowed to have parameters and/or return values of type sequence. Similarly, `SEQUENCE PROCEDURE` would be used to define procedures operating on sequences.

Meta Sequence Functions - `ENUMERATES`, `REDUCES`, etc. would be provided as built-in functions. As in the Lisp implementation, these would appear syntactically to be functions taking functional arguments; however, they would be handled by the compiler essentially as macros.

Loop Expression Blocks - A new keyword BEGIN SEQUENCE EXPRESSION would be introduced. This could be used in place of BEGIN in begin blocks, subprogram bodies etc. Only these blocks would be allowed to have variables of type sequence. Each such block would be compiled into a single loop.

Coercions - Given that the sequence data type would be used to identify all of the variables which carry sequences, various coercions could be supported in exactly the same way as in the Lisp implementation.

The following examples show what loop expressions would look like in Ada. The first is a program which takes in a vector of digits and computes the corresponding integer. The second program illustrates the definition of a sequence function.

```
type int_vector is array (integer range <>) of integer;
type int_sequence is sequence of integer;

function digits_to_number_ada(digits: int_vector) return integer;
  function times_ten(x: integer) return integer;
    begin return x*10; end;
  digit, scale: int_sequence;
begin sequence expression
  digit := Evector(digits);
  scale := generateS(times_ten, 1);
  return Rsum(digit*scale);
end;

sequence function Rsum(ints: int_sequence) return integer;
begin sequence expression
  return reduceS(+, 0, ints);
end;
```

Due to the type information which has to be specified and the fact that there is no compact representation for literal functions, the above programs are quite verbose. However, they are identical in basic structure to their Lisp counterparts.

IV - Comparison With Other Loop Notations

Consider the program SUM-POSITIVE-EXPRESSIONAL (reproduced below) which was used as an example in the beginning of this paper. There are many different computationally equivalent ways to represent any given loop. All of these representations are capable of expressing the same basic looping algorithm. In order to evaluate the usefulness of these representations, we must look at other characteristics beyond expressiveness.

```
(defun sum-positive-expressional (vector)
  (Rsum (Fgreater (Evector vector))))
```

The paramount property required of a looping representation is *understandability* i.e., how easy is it to look at a loop and determine what the loop is computing. Two closely related properties are also of great importance. The first is *constructibility* i.e., given a specification, how easy is it to build up a loop which satisfies the specification. The second is *modifiability* i.e., given a loop, how easy is it to change it in accordance with a change in its specification.

The key idea behind the expressional loop notation is that most looping algorithms are built up out of stereotyped fragments of looping behavior and therefore loop programs are easier to understand, construct, and modify if these fragments are expressed as easily identifiable syntactic units. In the expressional notation, loop fragments are represented by sequence functions. Many other looping notations have methods for representing at least some loop fragments. Discussion of these methods is the major theme of the comparisons below.

Two things act as the focus for the following sections. The first is the loop in the program SUM-POSITIVE-EXPRESSIONAL. Each section shows how the loop notation being discussed could be used to express this algorithm. The second focus is the six basic features of the expressional notation. The sections are ordered from simple constructs which have very few of these features to languages like APL and Hibol which embody most of them.

PROG and GO

The program SUM-POSITIVE-GO shows how our example loop could be implemented using a PROG and GO. The program is not very easy to understand because PROG and GO suggests a particularly unfortunate way to think about the loop, namely that it is basically a straight line piece of code which is converted into a loop by the addition of a GO. This notation embodies none of the basic features of the expressional notation. The key idea which is being missed by this way of thinking is that straightforward loops like this one are built up out of standard fragments of loops and not out of standard straight line fragments.

```
(defun sum-positive-go (vector)
  (prog (sum i end)
        (setq sum 0)
        (setq i 0)
        (setq end (1- (array-length vector)))
      L (cond ((> i end) (return sum)))
        (cond ((plusp (aref vector i))
               (setq sum (+ sum (aref vector i)))))
        (setq i (1+ i))
        (go L)))
```

Instead of highlighting the loop fragments, the program breaks them up into pieces and then mixes the pieces together. For example, the enumerator is broken up into three pieces: an initialization which sets the starting value for I, a termination test that terminates the loop after the last index is produced, and a repetitive step which increments I each time around the loop.

; just as difficult to see how the fragments interact as it is to identify the fragments themselves. The rator and the filter interact by sharing the variable I. In contrast, the interaction between the filter and lucer is represented by embedding part of the reducer inside of the filter COND. This is particularly ing because the COND looks like it is implementing an ordinary straight line conditional fragment. One look carefully at the surrounding context in order to see that this is not the case.

hough the above points have been presented primarily as problems of understandability, they cause much trouble with regard to constructibility and modifiability. In particular, the fact that the nts are not localized means that neither the construction nor modification processes can be localized. reatly complicates both tasks. Another problem is that since the various fragments are just mixed r, there is no support for keeping them semantically separate. One must be particularly careful that ıcing a new fragment will not disturb one of the other fragments.

ther kind of problem with PROG and GO as a notation for straightforward loops is that it supports a r of features which are needed only in complex situations and which obscure simple loops by cluttering p. Two examples of these are: the fact that PROG supports multiple tags, GOs and RETURNs; and the fact llows multiple assignments to the key variables involved. These features are particularly problematical e even when they are not being used, you have to look very closely in order to determine that they are not being used. In the example, you have to verify that there is only one tag, one GO, and one RETURN t there is only one assignment to each of the critical variables in the loop before you can have any nce in what is going on.

re are algorithms for which a PROG and GOs are particularly appropriate. For example, if a program ents a finite state automaton, GOs can be used to directly model the transitions. GOs can also be used to ent various exotic multiple entry and multiple exit loops. However, it is generally recognized that GOs er the best way to implement simple loops.

Tail Recursive Style

program SUM-POSITIVE-RECURSIVE is written in tail recursive style. Though it looks very different M-POSITIVE-GO it specifies essentially exactly the same algorithm. A compiler which knew about tail n would produce the same object code for the two programs. SUM-POSITIVE-RECURSIVE is at easier to understand because much of the verbiage is removed. There is no longer any possibility iple tags, GOs, or RETURNs. As a result, the reader does not have to worry about them. In addition, the t each value changes only once on each cycle of the loop is easy to see.

```
efun sum-positive-recursive (vector)
(sum-positive-recursive1 vector 0 0 (1- (array-length vector))))

efun sum-positive-recursive1 (vector sum i end)
(cond ((> i end) sum)
      (T (sum-positive-recursive1 vector
                                  (cond ((plusp (aref vector i))
                                         (+ sum (aref vector i)))
                                        (T sum))
                                  (1+ i)
                                  end))))
```

PROG, the tail recursive style suggests a particular way of looking at a loop. Namely, that we should ze the task at hand into a problem that can be recursively reduced a step at a time to a problem that is o solve. In this case the trivial problem is adding up the positive elements of a sub-vector of length he generalized problem is adding up the positive elements of a sub-vector and adding this to an initial um. The recursive step involves adding one element into the partial sum, and reducing the size of the

sub-vector.

There are loops which can be best understood by looking at them from the recursive viewpoint. However, this program is not one of them. The problem is that the tail recursive style is no better than a PROG at highlighting the fragments that the loop is composed of. As above, the fragments are broken up and mixed together. In addition, the way the fragments interact is still unclear. For example, part of the reducer is still nested in the filter. The "(T SUM)" clause which has to be added into the filter COND makes that interaction even less clear than in the PROG above. Like PROG and GO, the tail recursive style does not support any of the features of the expressional notation.

FOR

The next few sections describe notations which begin to support the idea of a sequence function (i.e., fragments of looping behavior as identifiable units). They do not however support any of the other features of the expressional notation.

Most algorithmic languages have looping constructs which facilitate the construction of simple loops. A typical example of these is the Ada FOR construct [1]. The Ada program SUM_POSITIVE_FOR illustrates the use of this construct. One benefit of FOR is that like the tail recursive style, it clearly delimits the extent of the loop and makes it clear that there is no exotic control flow going on in conjunction with the loop. Unfortunately, it is less helpful with regard to the data flow. There is no easy indication that each of the critical variables is only modified once.

```
type int_vector is array (integer range <>) of integer;

function sum_positive_for(vector: int_vector) returns integer is
  sum: integer;
begin
  sum := 0;
  for i in vector'range loop
      if vector(i)>0 then
         sum := sum+vector(i);
      end if;
  end loop;
  return sum;
end;
```

A much more interesting aspect of FOR is that it explicitly represents one of the fragments -- the enumerator of integers over the bounds of the array. This explicit representation of the fragment is particularly useful because (unlike the FOR constructs in most other languages) the Ada FOR construct protects the semantic integrity of the fragment by prohibiting the loop counter from being modified inside the loop. As a result, this particular fragment is easy to understand, construct, and modify.

Unfortunately, FOR is only capable of supporting this one kind of enumerator. There is no support at all for any of the other fragments in the loop. They are represented using straight line code in exactly the same way as in SUM-POSITIVE-GO. As a result, FOR is really not that much of an improvement over GO with regard to these other fragments.

Iterators in CLU

The language CLU [8] has extended the concept behind the FOR construct so that it can represent other enumerators besides integer enumeration. In CLU you can define a program called an *iterator* which takes in some unitary arguments and creates a sequence of objects. The iterator can then be used in a FOR in order to enumerate a sequence of elements to be processed in the body of the FOR. CLU provides a number of standard iterators including one corresponding to EVECTOR. As an illustration, the first program below shows how EVECTOR could be defined if it did not already exist. The program SUM_POSITIVE_CLU then shows how the iterator could be used.

```
Evector = iter(a: array[int]) yields(int)
  i: int := array[int]$low(a)
  end: int := array[int]$high(a)
  while i <= end do
    yield(a[i])
    i := i + 1
  end
end Evector

sum_positive_clu = proc(a: array[int]) returns(int)
    sum: int := 0
    for e: int in Evector(a) do
        if e > 0 then sum := sum + e end
    end
    return(sum)
end sum_positive_clu
```

Because there are no restrictions on the form that the body of an iterator can take (for example there is no requirement that it even be a loop), iterators are more general than the enumerators presented here. However, this power has drawbacks. For example, it makes it more difficult to define a meta sequence function like ENUMERATES. In addition, it would be difficult to treat iterators like macros and compile them inline in the loops which use them. The current CLU compiler implements iterators as separate procedures which return one element of the sequence every time they are called.

From the standpoint of understandability, an important aspect of iterators is that their semantic integrity is protected by the fact that they encapsulate their own state. In fact, iterators embody the logical concept of enumeration fully as well as the enumerators presented here. (It should be noted that the language Alphard [14] has a similar construct called a *generator*.) Unfortunately, neither of these languages provided any support for any fragments other than enumerators. As a result, each of these constructs is only a limited (though significant) improvement over simple FOR in the direction of supporting fragments.

Lisp DO

Another variant on FOR is the Lisp DO construct. This construct is interesting because it recognizes the existence of loop fragments other than enumerators and attempts to group their parts more closely together. Each of the initial lines of the DO is capable of representing a loop fragment. For example, the initialization and repetitive step of the index enumerator are combined together in the first line of the DO in the program SUM-POSITIVE-DO.

```
(defun sum-positive-do (vector)
  (do ((i 0 (1+ i))
       (end (1- (array-length vector)))
       (sum 0))
      ((> i end) sum)
    (cond ((plusp (aref vector i))
           (setq sum (+ sum (aref vector i)))))))
```

Unfortunately, DO is very restrictive in the way it can represent fragments. For example, the termination test of the enumerator has to be specified separately, causing the enumerator to be less conveniently represented than in a FOR. In addition, there is no good way to represent a filter at all. Going beyond this, the interactions between the fragments have to be represented in the same clumsy ways as in the programs above. For example, a COND still has to be used to express the interaction between the filter and the reducer.

At a more fundamental level, although DO makes it easier to write loop fragments as identifiable units, it does not enforce their semantic integrity. For example, you could easily put an assignment to I in the body of the DO. If you did this the computation involving I would no longer be an index enumeration. This would be particularly confusing because the first line of the DO would still look like an ordinary index enumeration.

All in all, it is clear that the various FOR and DO constructs are quite beneficial because they make it easier to locate a simple loop, and to verify that it is indeed simple. However, although these constructs point in the direction of explicitly supporting loop fragments they do not do this in either a very thorough way or a very semantically strong way. As a result, they are only a modest help in the understanding, construction, and modification of loops.

The Lisp Map Functions

The Lisp MAP functions are very restricted in what they can do. For example, they cannot be used to express the algorithm used in the examples above. However, when they can be used they are very compact and easy to understand. Each of the six MAP functions is an abbreviation for a particular combination of loop fragments. The pair of equivalent expressions below shows the fragments corresponding to MAPCAR.

```
        (mapcar #'function x)
same as: (Rlist (mapS #'function (Elist x)))
```

Each MAP function embodies a certain set of fragments and protects their semantic integrity. If these fragments are appropriate to the algorithm at hand, then the use of the MAP function leads to a program which is easy to understand, construct, and modify. The expressional loop notation is designed to extend the basic idea embodied in the MAP functions to a wider domain of programming.

The Lisp Macro LOOP

The Lisp macro LOOP [3] is a significant improvement over the constructs presented above because it recognizes loop fragments of all kinds as full fledged constituents. Consider the program SUM-POSITIVE-LOOP. In this program, the enumerator, filter, and reducer are each represented on a separate line in the loop. This gives a program which is much easier to understand, construct, and modify than the ones above. A number of standard loop fragments are supplied as part of the macro.

```
(defun sum-positive-loop (vector)
  (loop for item being each vector-element of vector
        when (plusp item)
        sum item))
```

In addition to supporting relatively general fragments and their combination, LOOP supports the creation of user defined fragments of all kinds. The example below shows how one could define VECTOR-ELEMENT OF which is the equivalent of the sequence function EVECTOR.

```
(define-loop-path vector-element Evector (of))

(defun Evector (ignore variable ignore phrases ignore ignore ignore)
  (sublis (list (cons 'expr (cadar phrases))
                (cons 'variable variable)
                (cons 'vector (gensym))
                (cons 'i (gensym))
                (cons 'end (gensym)))
          '(((vector) (i 0) (end))
            ((setq vector expr)
             (setq end (1- (array-length vector))))
            (> i end)
            (variable (aref vector i))
            nil
            (i (1+ i)))))
```

Unfortunately, LOOP neither develops the concept of a sequence, nor the analogy of treating loop fragments as functions. This prevents it from expressing loops as sequence expressions in analogy with ordinary unitary expressions. Instead, LOOP supports a keyword-based syntax which specifies both the fragments to be used, and how they are combined. The way fragments can be combined is rather restricted because it is tied up with the keyword parsing algorithm.

In addition, the LOOP macro has a body part (not used in the example above) just like the body of a DO. This body can contain arbitrary computation -- there is no attempt to protect the semantic integrity of the individual fragments in the initial part of the LOOP.

Another problem with LOOP is that the facilities it provides for defining the equivalent of new sequence functions are rather cumbersome. Unlike the expressional notation, there is nothing corresponding to the meta sequence functions. The user has to define a function which can deal with parsing parts of the LOOP syntax and which returns a list of six pieces which are put in different places in the loop being constructed. Acting together, these pieces have to perform the actions of the desired sequence function. At the most basic level, this is quite similar to what happens in the expressional notation. However, it seems better if the user does not have to interact with the system at this low a level.

APL

There are several programming languages which support what are essentially expressional loop notations. The oldest of these is APL [10]. It is interesting to note that there is no reason to believe that the developers of APL had anything like the expressional loop notation in mind. Rather, they were just seeking to provide a set of very useful operations on arrays. However, a style of writing APL has evolved where sequences are implemented as arrays.

The implementation of sequences as bona fide data objects automatically supports four of the six features of the expressional notation (i.e., sequences, sequence functions, user definition of sequence functions, and loop expression blocks). As illustrated below, both sequence functions and the vector summing algorithm can be very compactly represented in APL. Note that since sequences are directly represented as vectors, there is

no need for the function EVECTOR.

```
     ∇ RESULT←FGREATER VECTOR
[1]    RESULT←(VECTOR>0)/VECTOR
     ∇

     ∇ RESULT←RSUM VECTOR
[1]    RESULT←+/VECTOR
     ∇

     ∇ SUM←SUMPOSITIVEAPL VECTOR
[1]    SUM←RSUM(FGREATER(VECTOR))
     ∇
```

APL also has operators similar to the meta sequence functions. For example, "*function*/*value*" is the same as (REDUCES *function init value*) and "*function**value*" is the same as (SCANS *function init value*). (In both cases the init is automatically chosen to be the identity element under *function*.) Unfortunately, user defined functions cannot be used with these operators, so each one only actually corresponds to a small number of built-in sequence functions. APL supports the notion of a filter in a more general way than it supports REDUCES and SCANS. "(*function*(*value*))/*value*" is the same as (FILTERS *function value*). This operator (the two argument form of /), which is called compression, takes in two vectors and creates a vector of elements from the second vector which correspond to non-zero elements of the first vector. Any arbitrary function can be used to create the first vector. A binary function rather than a unary one is used in the example. (Note that compression makes a shorter vector, rather than introducing empty elements.)

APL has no operators corresponding to the meta sequence functions GENERATES, ENUMERATES, or TRUNCATES. Since sequences are represented as arrays, there does not have to be any equivalent of the sequence functions EVECTOR and RVECTOR. Further, since arrays are the only composite data structure supported by APL, there do not have to be any enumerators or reducers which deal with other data structures. Since all arrays are finite, there need not be any generators or truncators. APL does have an operator (the index generator "⍳*N*") corresponding to (ERANGE 1 N). Note that the fact that the meta operators provided by APL are somewhat limited does not prevent the user from defining any kind of sequence function he desires by simply using more primitive constructs to write the appropriate function on arrays.

APL also supports the idea of implicit MAPS to some extent. Every scalar function can be applied to vectors with the meaning that the operation is to be applied to every element of the vector. Also, scalars are coerced to vectors wherever necessary. Both of these processes are happening in the expression (*VECTOR*>0) above which takes in VECTOR and produces a vector of zeros and ones which indicate which elements of VECTOR are greater then zero. This cannot be done as completely as with the expressional notation presented here because there is no mechanism for differentiating between arrays which are arrays, and ones which are intended to be sequences.

There are two ways in which APL is more powerful than the expressional notation presented here. First, it supports a number of operators which are much more powerful. For example, it has a number of operators which rearrange the order and structure of an array such as reshape, concatenation (of two vectors), expansion (the inverse of compression), reversal, rotation, and grade up (sort). It has complex meta operators on pairs of arrays such as outer product and inner product which produce outputs which are not the same shape as the inputs. In addition to all this, arrays are of course also just data objects, and you can operate on them as such. You can retrieve and set individual elements and perform arbitrary computations.

Another way in which APL is more powerful is that while sequences are analogous to vectors, the standard intermediate structure in APL is the array. The fact that arrays are multidimensional makes them a more flexible representation. All of the operators above can be applied to arrays, and to selected parts of arrays producing results of similar or dissimilar shape.

The powerful features provided by APL make it possible to compactly express a wide variety of complex mathematical algorithms which cannot be expressed in the expressional notation at all. For these algorithms, APL has the virtue of easy understandability, constructibility, and modifiability. Unfortunately APL has several drawbacks. First, it does not support any data structures other than numbers, characters, and arrays. Second, although APL supports the expressional metaphor almost completely, it does not support the element at a time metaphor at all. Third, due to that fact that it supports such complex array functions and the fact that it rejects the element at a time metaphor, APL cannot in general be compiled into efficient code. Fourth, APL's approach to loops is embedded into a somewhat cryptic and forbidding syntax. Together, these features have limited APL's impact.

The expressional loop notation presented here eliminates these problems. First, it can handle arbitrary data structures. For example, to deal with a new aggregate structure, the user need only define new enumerators and reducers to convert the aggregate to a sequence and vice versa. Second the element at a time metaphor is part of the basis for the notation. Third, the expressional loop notation deliberately omits all those operations on sequences which would make it hard to compile. Fourth, the expressional notation is designed to be added into preexisting languages as a natural extension of their syntax. One need not learn a new language and environment in order to use it.

The Listless Transformer

In a Lisp-like language one could decide to support the expressional metaphor by implementing sequences as lists. Wadler [15] has implemented an interesting prototype system (the listless transformer) which is capable of transforming programs containing sequences implemented as lists and eliminating the actual computation of intermediate lists. The loop notation supported by his system is at heart essentially identical to APL with lists substituted for arrays. The target of his system is a Lisp-like language called Iswim [15]. The example below shows how sequence functions can be defined and used in this language.

```
def Evector(v) =
  Evector1(v, 0, length(v))
    where rec Evector1(v, i, end) =
      if i>end then nil
               else cons(aref(v, i), Evector1(v, i+1, end))

def rec Fgreater(xs) =
  case xs of
    nil => nil
    cons(x ,rest) => if x>0 then cons(x, Fgreater(rest))
                            else Fgreater(rest)

def Rsum(xs) =
  Rsum1(xs, 0)
    where rec Rsum1(xs, total) =
      case xs of
        nil => total
        cons(x, rest) => Rsum1(rest, total+x)

def sum-positive-listless(v) =
  Rsum(Fgreater(Evector(v)))
```

It is not clear whether any meta sequence functions are supported; however, they would be easy to implement as macros. In any case, the user can implement any sequence function he desires be defining arbitrary functions on lists. Iswim is a typed language and coercions like implicit introduction of MAPS can be supported.

Like APL, Wadler's notation is more powerful than the notation described here in that it supports

arbitrarily complex sequence functions and sequences can be multi-dimensional sequences of sequences. It goes beyond APL in being able to deal with arbitrary data structures.

Wadler's notation also shares APL's greatest weaknesses. It does not support the element at a time metaphor. In addition, due to the fact that arbitrarily complex sequence functions are allowed, it cannot be efficiently compiled in the general case. It also shares the problem that, since it is not oriented toward the element at a time metaphor, loops involving side-effects cannot be efficiently compiled.

Coroutines

Another language which supports most of the expressional metaphor is the coroutine language of Kahn and MacQueen [7]. They have suggested using parallel processes (coroutines) in order to represent computations communicating via one way *channels* (sequences). In their approach, unbounded sequences are implemented as real data objects which are passed an element at a time through channels between processes executed in parallel. The code below shows one way the vector summing algorithm could be implemented in their system. Each sequence function is defined as a separate process. These processes can have ordinary (unitary) inputs (e.g., the `VECTOR` input of `EVECTOR`) and outputs (e.g., the return value of `RSUM`). They can also have channel (sequence) inputs (e.g., the `CHANNEL1` argument of `FGREATER`) and outputs (e.g., the `CHANNEL` output of `EVECTOR`). An element is retrieved from a channel by the function (`GET` *channel*). An element can be put into a channel by the function (`PUT` *item channel*). In order to use the sequence functions, they are combined together in an expression as in the function `SUM-POSITIVE-COROUTINE`. This expression is placed in a `DOCO` form which causes the three processes to be executed concurrently.

```
process Evector vector => channel;
  vars i;
  1 -> i;
  repeat
     put(i, channel);
     increment i;
  until i>upper-bound(vector);
  put(done, channel)
endprocess;

process Fgreater in channel1 => channel2;
  vars n;
  repeat
    get(channel1) -> n;
    if n=done or n>0 then put(n, channel2) close
  until n=done
endprocess;

process Rsum in channel => sum;
  vars n, sum;
  0 -> sum;
  repeat
    get(channel) -> n;
    if not(n=done) then sum+n -> sum close
  until n=done;
  return(sum)
endprocess;

process sum-positive-coroutine vector
  start doco Rsum(fgreater(Evector vector)) closeco
endprocess;
```

The language of Kahn and MacQueen supports neither meta sequence functions, nor automatic coercions. However, they could be added if desired. It is interesting to note that, unlike APL, coroutines directly

support the element at a time metaphor.

The coroutine approach is more powerful than the expressional notation along a different dimension from APL. Each process is a truly independent parallel process. One aspect of this is that sequences can really be infinite. In addition, it is possible for one process to terminate without forcing the other ones to terminate and processes can dynamically spawn whole networks of other processes. This makes it possible to express modes of computation which cannot be conveniently expressed with any of the other notations discussed here. However, this brings with it a certain overhead. In the example above, the special token `DONE` is passed around between the processes so that the termination of the `EVECTOR` process will trigger the termination of the other processes.

The key drawback of the coroutine approach is that it is not clear how it can be compiled. Since it supports the element at a time metaphor, certain logical obstacles are removed; however, like APL, it supports the definition of arbitrarily complex sequence functions. Going beyond this, given that the coroutine notation is capable of expressing arbitrary parallel computations, one would expect that it will be extremely difficult to write an optimizing compiler which reliably detects groups of processes which interact merely as simple loops. However, without such a compiler, the coroutines impose an unacceptable overhead on the execution of simple loops.

The expressional loop notation presented here is based on ideas very similar to the coroutine notation; however it is restricted so that it is trivial to compile. The intention is to use the expressional notation to represent simple loops while reserving the coroutine notation for those situations where its greater power is required.

Hibol & Model

The language Hibol [12,13] is the oldest language which both supports the idea of a sequence and is completely compilable. It is a very high level business data processing language based on the concept of a flow (which is basically equivalent to a sequence). It is very strongly oriented toward the element at a time metaphor and relies heavily on the concept of the implicit introduction of `MAPS`. The body of each Hibol program is a nonprocedural set of expressions specifying the computations on typical sequence elements.

The program `SUM_POSITIVE_HIBOL` computes the sum of the positive elements in a file. (The only aggregate data type supported by Hibol is a file.) The language provides a few standard sequence functions (e.g., the operator `SUM` in the program below). In addition, the operator `IF` implements the meta sequence function `FILTERS`. These facilities make it possible to specify the body of `SUM_POSITIVE_HIBOL` as a simple expression. The `DATA DIVISION` part of the program describes the files accessed by the program in a format very similar to Cobol.

```
/* the program sum_positive_hibol */
data division
  key section
    key index
        field type is integer
  input section
    file vector_item
         key is index
         type is integer
  output section
    file sum_positive
         type is integer
computation division
  sum_positive is (sum of (vector_item if vector_item > 0))
```

Hibol is more powerful than the expressional notation in that flows are multidimensional objects like arrays where each level of index is an alphanumeric key rather than a number. The Hibol operators can be selectively applied to specific dimensions of a flow. A set of defaulting mechanisms make it possible to specify a simple program like the one above without having to explicitly specify which dimensions operators are being applied to. However, the operations which can be applied to flows have been carefully selected so that all flow expressions can be compiled into efficient loop code.

The Hibol compiler clearly shows that an expressional looping notation can be straightforwardly compiled even if it supports multidimensional sequences. Nevertheless, when designing the expressional loop notation presented here it was decided to omit this feature for two reasons. First, it was judged that the frequency of its use would not justify the extra complexity of supporting it. Second, when a loop algorithm becomes complex enough that the user is forced to specify which dimensions operators are being applied to, the syntactic mechanisms required cause the resulting expressions to begin to lose the virtue of easy understandability.

From the point of view of this discussion the primary weakness of Hibol is that it does not provide very much support for the expressional metaphor. First, it provides a few built in sequence functions, but does not allow the user to define new ones. Note that files are the only aggregate data structure supported by Hibol, and that enumeration and reduction of files occurs implicitly. Second, it supports only two meta sequence functions: `MAPS` (introduced only implicitly) and `FILTERS` (the form `IF`). Implicit nested loops can be specified but there is no notion of an explicit looping block.

As discussed above, the expressional loop notation presented here addresses these problems because it can deal with arbitrary data structures, because it supports the creation of user defined sequence functions, and because it is intended to be embedded in a language which supports standard control flow constructs. The expressional notation being presented here could be looked at as taking some of the key ideas embodied in Hibol and separating them out from the business data processing language context of Hibol in a form in which they can be conveniently added into other languages.

More recently, another language has been developed which is very much like Hibol. This language (Model [11]) is based on the same idea of a multidimensional sequence, and is also primarily intended for business data processing applications. It is somewhat more powerful, and has a somewhat wider range of features, but at the level of this discussion it is essentially identical to Hibol. It is fully compilable and has the same basic advantages and disadvantages. It serves as yet another example that the idea of a sequence appears in many different forms in many different languages.

References

[1] J.G.P. Barnes, "Programming in Ada", Addison-Wesley, London, 1982.

[2] T.A. Budd, "An APL Compiler", Univ. of Arizona, Dept. of Comp. Sci. TR 81-17, October 1981.

[3] G. Burke and D. Moon, "Loop Iteration Macro", MIT/LCS/TM-169, July 1980.

[4] D.P. Friedman and D.S. Wise, "CONS Should Not Evaluate Its Arguments", Indiana Tech. Rep. 44, Nov. 1975.

[5] L.J. Guibas and D.K. Wyatt, "Compilation and Delayed Evaluation in API.", in Proc. 5th ACM POPL Conf., Sept, 1978.

[6] P. Henderson and J.H. Morris, "A Lazy Evaluator, presented at the SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Atlanta, Jan. 1976.

[7] G. Kahn and D.B. MacQueen, "Coroutines and Networks of Parallel Processes", in 1977 Proc. IFIP congress, North-Holland, Amsterdam The Netherlands, 1977.

[8] B.H. Liskov, et. al., "CLU Reference Manual", Lecture Notes in Computer Science, G. Goos and J. Hartmanis editors, V114 Springer-Verlag, New York, 1981.

[9] D.A. Moon, "MacLisp Reference Manual", MIT Cambridge MA, April 1974.

[10] R.P. Polivka and S. Pakin, "APL: The Language and Its Usage", Prentice-Hall, Englewood Cliffs NJ, 1975.

[11] N.S. Prywes, A. Pnueli, and S. Shastry, "Use of a Non-Procedural Specification Language and Associated Program Generator in Software Development", ACM TOPLAS, V1 #2, October 1979, pp 196-217.

[12] G.R. Ruth, "Data Driven Loops", MIT/LCS/TR-244, 1981.

[13] G.R. Ruth, S. Alter, and W. Martin, "A Very High Level Language for Business Data Processing", MIT/LCS/TR-254, 1981.

[14] M. Shaw and W.A. Wulf, "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", CACM V20 pp 553-564, Aug. 1977.

[15] P. Wadler, "Listlessness is Better than Laziness", PhD Thesis, Carnegie-Mellon Univ., (to appear).

[16] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, Dec. 1978.

[17] R.C. Waters, "A Method for Analyzing Loop Programs", IEEE Trans. on Soft. Eng., V5 #3, May 1979.

[18] D. Weinreb and D. Moon, "Lisp Machine Manual", MIT Cambridge MA, July 1981.

Appendix A: The Compilation Process

The first section in this appendix describes some assumptions which the macro expansion process makes about the form of the loop expressions to be processed. The user must be careful to ensure that these assumptions are satisfied. The rest of the sections discusses the actual macro expansion process in detail. This discussion is intended to function both as detailed documentation for the actual program, and as a guide to anyone who wishes to implement a similar system.

The compilation process revolves around a data structure representing the key information about a fragment of looping behavior. Each fragment data structure contains all of the information needed to create a loop corresponding to a sequence function, and information about the inputs and outputs of the sequence function. A call on a sequence function is represented as an application of a fragment data structure to a list of arguments. The process of combining several sequence functions together into a single loop proceeds by combining together the fragment data structures corresponding to them.

Given a program that contains one or more loop expressions, macro expansion will proceed normally until the outermost macro in one of these loop expressions is encountered. At that time, the LETS macro package immediately locates all of the inner loop macros in the expression and constructs a loop combining them all together. Macro expansion then continues normally until another loop expression is encountered.

The process of converting a loop expression into a loop occurs in several steps. After locating an expression, all of the calls on sequence functions in it are converted into applications of fragment data structures. Similarly, calls on meta sequence functions are converted into applications of fragment data structures built out of the functional arguments passed to the meta sequence functions.

Once everything has been reduced to an application, the result is parsed in order to located nested loops. Any nested loops are isolated and processed separately. A second phase of parsing then performs coercions such as the automatic introduction of MAPS and AT-END. The resulting group of applications is combined together into a single fragment data structure. This structure is then converted into a single loop.

As an example, the following shows the code which is produced for the loop in the program SUM-POSITIVE-EXPRESSIONAL.

```
        (Rsum (Fgreater (Evector vector)))
becomes:(prog T (i end num sum)
                (setq i 0)
                (setq end (1- (array-length vector)))
                (setq sum 0)
            L   (cond ((> i end) (go E)))
                (setq num (aref vector i))
                (cond ((> num 0) (setq sum (+ sum num))))
                (setq i (1+ i))
                (go L)
            E   (return-from T sum))
```

Interaction With Other Macros

There are two important (and unfortunate) restrictions on the way in which the LETS macro package can be used which stem from the fact that the package does not have any special knowledge of either system or user defined fexprs or macros. These restrictions could be removed if more knowledge was built into the compilation process.

The first restriction is that in a loop expression, *every list whose* CAR *is one of the loop macros must be a call on that macro.* The list will be macro expanded and combined into the loop. To avoid running afoul of this assumption, you should never use the name of one of these macros as the name of a variable. The only place where this restriction does not apply is inside of quoted lists.

The second restriction is that *for each variable name in the argument list of a* DEFUNS, *bound by* LETS, *or in the lambda list of a literal lambda expression passed to a meta sequence function, every occurrence of that symbol in the body must be an instance of a reference to that variable.* The function SUBST will be used to rename this variable when necessary to avoid name conflicts. The two main ways that trouble could arise is if you use a variable name which is the same as a function name, or if you rebind the variable name in some inner scope. Note that you cannot even use the variable name in a quoted list.

Another kind of restriction arises from the fact that the LETS macro package does all of its processing without expanding any other macros. As a result of this, *only lists whose* CARs *are one of the loop macros are allowed to expand into a loop fragment* (*as opposed to a complete loop*). In particular, an ordinary macro cannot expand into code which is supposed to be a loop fragment. This will not work because the macro will not be expanded until after the loop it is in has already been completely constructed. Note that it is all right for a macro to contain a complete loop expression which will be converted into a loop as a whole by itself. The appropriate way to make macros describing loop fragments is to use DEFUNS. For example, compare the following two definitions of a loop fragment which enumerates the CARs of the elements of a list. Only the second one will work.

```
(defmacro car-Elist-buggy (input)
  (list 'car (list 'Elist input)))

(defunS car-Elist (input)
  (car (Elist input)))
```

Another consequence of the fact that LETS does extensive processing before other macros are expanded is that *you cannot nest one of the expressional macros inside a call of a macro that looks inside of its argument.* For example, even assuming that you define a SETF property for ELIST, you cannot write "(SETF (ELIST L) X)". The problem is that since the loop macros are expanded first, SETF will never get to see the ELIST. Also note that instances of loop macros are usually replaced by variables in the resulting loop. However, you can say things like the following "(SETF (CAR (ELIST L)) X)" because the SETF does not need to look at the argument of the CAR.

The Representation for a Fragment

Loop fragments are represented internally by the following structure:

```
(S-frag name ((arg type . info) ...)
  icode code1 code2 pcode ucode)
```

The *name* part of the form is used for producing more understandable error messages. It records what macro generated the fragment. The second part of the form is a list of argument descriptors. The symbol *arg* is the name of the argument. Every internal use of the argument is represented by that symbol. There is one argument declaration for every input, output, and auxiliary variable used by the fragment. The order of the

declarations is used to match the inputs up with parameters when the fragment is used. The symbol *arg* is created by GENSYM. It is guaranteed to be unique and occur only in this single fragment. As a result, renaming does not have to be used when fragments are combined together. If the fragment is copied, then the *args* are renamed by using SUBST. As a result, every internal instance of the symbol must correspond to a use of the variable.

Note that free variable inputs and outputs are not mentioned in the argument descriptors. Rather, they are just referred to in the body of the fragment where appropriate. Due to the fact that the order of execution in a loop expression is preserved, things work out all right when fragments are combined together without the system having to take any explicit action. In fact, the system ignores the presence of free variables entirely.

Each argument has a *type* which is one of the following symbols: UI, SI, UO, SO, UP, SP, A, and F. These symbols are built up out of the following code letters. The use of some of the code letters requires that additional information be supplied in the *info* field.

U - UNITARY - This is an ordinary Lisp object. Inputs are given a value before the entire fragment is evaluated, and outputs pass out their final value when the whole evaluation is over.

S - SEQUENCE - This specifies that the argument is a sequence. The variable holds successive elements of this sequence. On each cycle of the loop, each sequence input is given a new value before the first time it is read, and the final value of each sequence output is exported out of the fragment.

I - INPUT - This is an input object passed in via nesting in argument position.

O - OUTPUT - This is an output passed out through the return value. There can be more than one return value in which case their order specifies which is which in a MULTIPLE-VALUE.

A - AUX - This is an internal auxiliary variable. It must be unitary. If the *info* field is non-NIL, it indicates that this variable was specified by the user and must be retained in the final loop produced.

P - OPTIONAL - This is an optional input. When the definition is applied it is converted into either an input or an aux. The *info* field contains an initializing expression to use when a parameter value is not supplied for this argument.

F - FLAG - This is an auxiliary flag used in filtered computations. The filtered sequences themselves are carried in separate variables. The *info* field is a list of all of the free variables and return values which are under the control of this filter flag.

The remainder of the fragment specifies the computation to be performed. The *icode* is a list of zero or more expressions which are executed exactly once just before the repetitive part of the loop is executed. The *icode* cannot refer to any sequence arguments. It can read only unitary inputs. It can write any aux, or unitary output. Its effect is to give initial values to variables. Typically, every unitary output is given some default value.

The *code1* and *code2* are the repetitive body of the fragment. They are the only places where sequence arguments can be referred to. Both of these are lists of zero or more expressions. Both of them are executed on every cycle of the loop and can read sequence values. The *code1* (but not the *code2*) can write sequence values.

There are two different slots here because of the following property. All the *code1* parts of all the fragments being used will be executed before all of the *code2* parts. This gives you control over what is going on. In particular all terminations are placed in *code1* parts. As a result, you can depend on the fact that the *code2* will not be executed on the cycle where the loop terminates. (The *code1* may be.) If there is a filter producing some of the inputs read by *code1* or by *code2* then both will be evaluated only on those cycles where all of the filtered inputs are available.

minations are represented by putting a COND with DONE in one of its branches in the *code1*. Filters are nted by using the form (S-IF *flag-test*. *actions*) in the *code1*. The effect of the filter is obtained by g sequence outputs in the actions and making the appropriate flag argument declarations.

pcode is a list of zero or more expressions which is executed exactly once after the loop, if it terminates ly. It cannot refer to any sequence quantities. Its purpose is to perform epilog computations involving lary outputs.

ucode is a list of zero or more expressions which is executed in an UNWIND-PROTECT wrapped around p eventually produced. It cannot refer to any sequence quantities. Its purpose is to perform epilog ations involving the unitary outputs which must be performed no matter how the loop is terminated.

following examples illustrate the fragment representation. The first corresponds to the sequence n RLIST. Note the use of some *pcode* in order to reverse the list CONSed up. The second corresponds NGE. Note the use of an optional parameter BY, the presence of a terminator, and that the entation of the state is placed in *code2* so that it will not be done on the cycle on which the loop ites. The final fragment corresponds to FGREATER. Notice that the computation of the filter flag is ed from the S-IF which specifies what values are controlled by that flag.

```
-frag 'Rlist ((item SI) (result UO))
((setq result nil))
((setq result (cons item result)))
()
((setq result (nreverse result)))
())

-frag 'Erange ((state UI) (end UI) (by UP . 1) (int SO))
()
((cond ((> state end) (done))) (setq int state))
((setq state (+ state by)))
()
())

-frag 'Fgreater ((int SI) (limit UP . 0) (flag F big) (big SO))
()
((setq flag (> int limit)) (S-if flag (setq big int)))
()
()
())
```

of the internal macro processing revolves around fragments represented in the above form. They are ed together into larger and larger fragments and then converted into normal loop code.

Sequence Functions

form (S-APPLY *outs fragment parameter...*) is used to apply a fragment to a group of parameters. field indicates what variables (if any) the outputs are being assigned to. T indicates that the outputs being returned. Sequence functions are merely macros which expand into S-APPLYs of fragments. lustrated by the following pair of expressions. Note that each time the fragment is instantiated, the ts in it are renamed so that there cannot be any name clashes.

```
      (Rlist x)
e as: (S-apply T (S-frag 'Rlist ...) x)
```

nly interesting thing which happens when a sequence function is expanded into an S-APPLY is the t of optional arguments. If a parameter is provided, then the argument list of the fragment is so that the argument is specified to be a normal input. If no parameter is supplied then the argument

converted into an aux and the initializing expression is used to give the argument a value either in the *icode* if it is unitary) or in the *codel* (if it is a sequence value). Note that this expression is evaluated inside the ragment and can refer to all of the arguments which precede it. The case of an optional argument not being upplied is illustrated below. Note the general form of this fragment shown above.

```
        (Erange 1 10)
same as: (S-apply (S-frag 'Erange ((state UI) (end UI) (by A) (int SO))
                    ((setq by 1))
                    ((cond ((> state end) (done))) (setq int state))
                    ((setq state (+ state by)))
                    ()
                    ())
                  1
                  10)
```

The way the S-APPLY forms created by sequence functions are themselves handled is discussed below.

Meta Sequence Functions

Like sequence functions, meta sequence functions produce S-APPLYs of fragments. However, unlike a equence function, some of the arguments to a meta sequence function are used to compute what the ragment should be. The meta sequence function REDUCES is used as an example in the pair of forms below. Note how the *name* field of the S-FRAG is used to record the initial meta sequence function expression.

```
        (ReduceS #'(lambda (a b c) body) init seq1 seq2)
same as: (S-apply (S-frag '(ReduceS #'(lambda (a b c) body) init seq1 seq2)
                          ((in UI) (b SI) (c SI) (a UO))
                    ((setq a in))
                    ((setq a body))
                    ()
                    ()
                    ())
                  init
                  seq1
                  seq2)
```

Note that since the variables of the literal LAMBDA become variables of the resulting fragment, they must e *unique in body*, because SUBST may be used to rename them. If a literal function name is used instead of a iteral LAMBDA, then it is converted into a LAMBDA.

Locating Loop Expressions

Before a loop expression can be processed, it has to be located in its entirety. There are three ways in hich this can happen. The easy case is when the loop is delimited by a LETS or DEFUNS. In that case there is o difficulty in identifying it.

The second case is also quite easy. Whenever any of the sequence functions or the meta sequence unctions is encountered unexpectedly (i.e., not during the processing of a loop which has already been ocated) it is wrapped in a LETS. Processing then continues as if the LETS had always been there.

The third process is much more complex. As soon as a loop expression is located by either of the above ethods, all of the sequence functions and meta sequence functions inside it are expanded into S-APPLYs. ny LETSes found inside are processed completely as subloops before processing continues on the outer loop. Note that no other macros are expanded at this stage or at any stage during the processing of loop xpressions.) Once this is done the result is parsed in order to detect nested loops. Once they are located, ey are wrapped in LETSes and immediately processed.

Nested loops are located by looking at the types of inputs and outputs to S-APPLYs inside a loop expression. These types are specified in the argument lists of the fragments the S-APPLYs are applying. To simplify the discussion assume for the moment that every function in a loop expression took exactly one argument. Every S-APPLY takes in either a unitary thing U or a sequence S and returns one of the two. They will be annotated as either U-U, U-S, S-U, or S-S.

Consider the examples below of how fragments can fit together. The first example shows a loop expression where everything fits together, and there is no nested loop. The second example shows the prototypical case of a nested loop. If the two inner fragments are grouped together into a subloop and then MAPSed over the stream coming out of the first fragment, then everything will work fine. Otherwise there is no good way to make things fit together. The search for nested loops focuses on finding balanced subexpressions which clash with their surroundings at both ends. The only real difficulty is that the expression as a whole may be unbalanced at either end (as shown in the final two examples) so there is no dependable place where parsing can start.

```
            (Rsum (Fgreater (Elist x)))
             U-S   S-S       S-U

            (Rlist (Rlist (Elist (Elist x))))
             U-S    U-S    S-U    S-U
parsed as:   U-S   <U-S    S-U>   S-U

            (Rlist (Elist (Elist x)))
             U-S    S-U    S-U
parsed as: <U-S    S-U>   S-U

            (Rlist (Rlist (Elist x)))
             U-S    U-S    S-U
parsed as:   U-S   <U-S    S-U>
```

Note that ordinary functions embedded in loop expressions are MAPSed if they end up receiving a sequence and otherwise are just executed normally. Therefore, they always return the same type that they receive and do not have to be considered when looking for nested loops. The real parsing algorithm has to work on tree-like expressions and is extended accordingly. It looks for balanced subtrees which clash with their surroundings at both root and fringe. They are then separated out as subloops.

LetS

One purpose of a LETS is to delineate a loop as discussed above. The other is to define sequence variables. All of the variable value pairs are simplified as shown below by putting the initializing expressions inside the LETS. Note that this means that these expressions cannot refer to the values which any of the bound variables have outside of the LETS. Destructuring is handled by expanding it into a group of SETQs.

```
        (lets ((x (Erange 1 10))
               ((a b) (Elist list)))
          (reverse (Rlist (list 'item x (+ a b)))))
becomes:(lets (x a b y)
          (setq x (Erange 1 10))
          (setq y (Elist list))
          (setq a (car y))
          (setq y (cdr y))
          (setq b (car y))
          (reverse (Rlist (list 'item x (+ a b)))))
```

Note that the only variables which carry sequences inside the body of the LETS are the ones specified in the bound variable list. All of the free variables referred to in the body are unitary no matter what they are in

the place where they are defined. This reflects the fact that if this loop is nested in another loop then it will be MAPSed and so any sequences in that loop will look like unitary values from its point of view. Note that if sequences were multidimensional objects as in APL then things would be much more complicated because each level of looping would only strip a single dimension off of a sequence.

Implicit MapS, and Coercions

The processing of the body of a LETS starts by breaking apart all of the S-APPLYs as follows. SETQs of new variables are created as needed so that every argument to an S-APPLY is a variable, and the output of every S-APPLY is immediately put in a variable. This transformation is illustrated below. Note that the output variable fields of the S-APPLYs are used to specify the destinations for their outputs. The use of a MULTIPLE-VALUE will lead to a list of more than one variable in this field. The types of the variables are chosen so that the types match the argument types of the fragments and so that the inputs and outputs to an ordinary expression are all of the same type. The fact that nested loops have already been removed guarantees that this is in fact possible. Note that execution order is preserved when expressions are broken apart.

```
          (lets (x a b y)
            (setq x (Erange 1 10))
            (setq y (Elist list))
            (setq a (car y))
            (setq y (cdr y))
            (setq b (car y))
            (reverse (Rlist (list 'item x (+ a b)))))
becomes:(let (u1 u2 u3)
          (lets (x a b y z)
            (setq u1 1)
            (setq u2 10)
            (S-apply (x) (S-frag 'Erange ...) u1 u2)
            (S-apply (y) (S-frag 'Elist ...) list)
            (setq a (car y))
            (setq y (cdr y))
            (setq b (car y))
            (setq z (list 'item x (+ a b)))
            (S-apply (u3) (S-frag 'Rlist ...) z)
            (reverse u3)))
becomes:(let (u1 u2 u3)
          (lets (x a b y z)
            (S-apply (u1) (S-frag '(at-start #'(lambda () 1)) ...))
            (S-apply (u2) (S-frag '(at-start #'(lambda () 10)) ...))
            (S-apply (x) (S-frag 'Erange ...) u1 u2)
            (S-apply (y) (S-frag 'Elist ...) list)
            (S-apply (a) (S-frag '(mapS #'car ...) ...) y)
            (S-apply (y) (S-frag '(mapS #'cdr ...) ...) y)
            (S-apply (b) (S-frag '(mapS #'car ...) ...) y)
            (S-apply (z) (S-frag '(mapS #'(lambda (m n o)
                                            (list m (+ n o))) ...) ...)
                         x a b)
            (S-apply (u3) (S-frag 'Rlist ...) z)
            (S-apply T (S-frag '(at-end #'reverse ...) ...) u3)))
```

While things are being decomposed, the following coercions are applied. Every ordinary expression which was nested in an S-APPLY which requires a unitary value is computed AT-START. Every ordinary expression which appeared at top level and which receives unitary output values from sequence functions is computed AT-END. Every other ordinary expression is MAPSed. RLAST is used to coerce the value of the last form to unitary if it isn't unitary already. These coercions lead to the third version of the LETS above. Note that the introduced meta sequence functions are immediately expanded into S-APPLYs.

Combining Fragments

Once all of the appropriate coercions have been applied, the fragments are combined together into one large fragment. This is done in two stages. First, each S-APPLY is converted into a naked fragment. Note that every argument to an S-APPLY is a variable, and its outputs go directly to variables. (The example below shows the RLIST in the example program above.) The S-APPLY is converted into a naked fragment by merely renaming the arguments to the appropriate variables and making them free as shown.

```
(S-apply (u3) (S-frag 'Rlist ((item SI) (result UO))
                      ((setq result nil))
                      ((setq result (cons item result)))
                      ()
                      ((setq result (nreverse result)))
                      ())
              z)
becomes:(S-frag ()
          ((setq u3 nil))
          ((setq u3 (cons z u3)))
          ()
          ((setq u3 (nreverse u3)))
          ())
```

Any outputs which are not used are converted to aux variables. After this phase, the only arguments which remain in fragments are aux and flag variables. The outputs of the last form in the LETS are retained as return values and will be used as the return values form the loop as a whole. Note that the last form may be a VALUES.

Now that everything is a fragment, they are all combined together starting at the top. This combination goes pairwise as shown below. The new fragment is created merely by concatenating the corresponding parts of the two initial fragments. As a result, the order of evaluation is preserved. Note that due to all of the renaming that occurred above, all of the data flow works out right without any special processing being necessary. Also since all variable names in the original fragments were GENSYMs there is no possibility of unintentional name clashes.

```
          ((S-FRAG argsa icodea codela code2a pcodea ucodea)
           (S-FRAG argsb icodeb codelb code2b pcodeb ucodeb))
becomes:(S-FRAG argsa-argsb
           icodea-icodeb
           codela-codelb
           code2a-code2b
           pcodea-pcodeb
           ucodea-ucodeb)
```

The only complexity is involved with filters. If any of the variables read by *codelb* or *code2b* are controlled by filter flags in the first fragment, then both *codelb* and *code2b* are nested in S-IF forms predicated on the AND of these flags. For example, suppose that *codelb* reads two sequence variables S1 and S2, which are controlled by the flags F1 and F2 respectively. In this case, *codelb* would be converted to (S-IF (AND F1 F2) . *codelb*) before combination. *Code2b* would be converted analogously.

The Form of the Loops Produced

Once all of the fragments have been combined into a single large fragment, this fragment is converted into a loop as indicated below. The various parts of the fragment are merely concatenated together into the body of a PROG. *Var-list* is a list of all of the aux, flag, and return variables which are specified in *arg-list*. The return-values are the return variables from *arg-list*. If the fragment contains any *ucode* then the PROG produced is wrapped in an UNWIND-PROTECT containing this *ucode*.

```
        (S-FRAG args icode (T code1 code2) pcode)
becomes:(PROG T var-list
           S   icode
           L   code1
               code2
               (go L)
           E   pcode
               (RETURN-FROM T . return-values))
```

Note that the PROG produced is just basic Lisp. (On the LispMachine this PROG is named T so that it will be transparent to the user.) The PROG contains a number of variables and tags created by the macros. These are all GENSYMs and so that they cannot conflict with any user variables. As a debugging feature, the macros make sure that all of the stream variables specified in a LETS become variables in the PROG. At a break point, you can look at these variables in order to see the current element in each of the corresponding sequences. Also for debugging convenience, the variable LETS:S-PROG holds the PROG produced from the most recently macro expanded loop expression. You can look at it in order to see exactly what code was produced.

The form (DONE) expands into (GO E). The form (DONE . *results*) expands into (RETURN . *results*). Note that no special action is taken with regard to terminations, they just end up in the right places as things are combined together. The form (RESTART) expands into (GO S). This also just ends up in the right place. The form (S-IF *pred* . *actions*) expands into (COND (*pred* . *actions*)).

DefunS

The purpose of a (DEFUNS *name lambda-list* . *body*) is to define a sequence function. The *body* is exactly like the body of a LETS. In addition the aux variables in the *lambda-list* are just like LETS variables. These variables and the *body* are processed exactly as described above in order to create a fragment. The arguments in the *lambda-list* specify that some of the free variables in the fragment are actually non-free inputs. The fragment is modified to reflect this. Note that these variables must be unique in the body so that the system can use SUBST to rename them. A sequence function macro is then constructed with the appropriate *name*.

Variable Simplification

One problem with the compilation process outlined above is that it creates a very large number of variables which end up not really doing anything useful. Due to the fact that the LispMachine compiler is not capable of optimizing away these variables, the macro package performs a set of simplifications in order to get rid of them itself.

The following simplifications are performed wherever possible. Note that this process is applied only to the variables created by the system. The variables explicitly declared in a LETS are never removed, and any free variables used in the loop are never removed.

1) If a variable is never read than it is eliminated. Any computations performed to assign values to it are also eliminated if it can be established that there is no possibility of a side-effect occurring.

2) If you have (SETQ X Y) and X is SETQed only once and Y is not SETQed in the range of reading X, then the two variables can be merged together into one variable eliminating whichever one can be eliminated.

3) If you have (SETQ X EXPR) and X is read only once, and nothing read by EXPR is modified between here and the use of X and there is no possibility of trouble with side-effects from moving EXPR, then EXPR can be substituted for X eliminating X.

Another area where needless complexity results is filters. The processing above leads to the use of a number of flags and S-IF forms. These are simplified as follows: If two S-IFs in a row are predicated on the same flag expression then they are combined together into one in order to reduce the number of references to the flags. When this is done in conjunction with the variable simplifications above, simple cases of filters end up as just simple CONDs.

Appendix B: Functional Summary

This Appendix is intended as a short reference manual for the system. It assumes that you have already read the rest of the paper and just gives a very brief description of each of the macros available to the user. The macros are listed in logical groupings. Note that all of these macro names are global on the LispMachine. The summary begins with a description of the basic macros.

letS ((*var value*) ...) **&rest** *body*

This has two purposes: to define a group of variables which contain sequences of values, and to indicate that a group of sequence expressions (the *body*) should be combined together into a single loop. Each *value* will be coerced to a sequence. If it is omitted (or if the var-value pair is rendered as merely a symbol) then the initial value is undefined and the variable must be written before it can be read. A tree of *vars* instead of a symbol can be specified, in which case destructuring is performed. Note that every free variable is per force unitary.

All of the expressions in the body are combined into a single loop. Each unitary expression in the body will be automatically MAPSed if possible. The only time it is not possible is if it uses the output of some reducer. In this latter case, the expression will be automatically computed AT-END. The value of the last expression in the *body* is coerced to unitary and returned as the value of the loop.

In the body, you can use SETQ to assign to a sequence variable. MULTIPLE-VALUE can be used to access the multiple values of a sequence function. The last form can be a VALUES indicating that multiple values are to be returned from the loop as a whole.

defunS *name lambda-list* **&rest** *body*

The purpose of this form is to define a new sequence function. The *lambda-list* is just like an ordinary lambda list except that it supports only the following four keywords. &UNITARY indicates that following arguments are unitary. This is the default to start with. &SEQUENCE indicates that the following arguments carry sequences. &OPTIONAL indicates that the following arguments are optional. &AUX indicates that the following arguments are internal variables. With both of the last two cases default values can be specified by rendering the argument as a variable-value pair. If no default value is specified then the value the variable is undefined, and the variable must be written before it can be read.

DEFUNS defines a macro of the specified *name* defining the sequence function specified by *body*. The *body* is exactly like the body of a LETS except that it is not immediately coded up into a loop, and the value of the last expression is not coerced to unitary. Rather, this value is returned whether it is unitary or a sequence.

done &rest *results*

In a loop expression the macro DONE can be executed in order to indicate that the loop should be immediately terminated. If no *results* are specified, then the loop will be terminated normally executing all AT-END code, and returning the result specified by the last expression. If any *result* arguments are supplied then they will be returned as the values of the loop. Note, however, that in this case any AT-END code will be skipped. Any AT-UNWIND code is executed in either case.

restart

Executing this inside of a loop expression causes the immediately containing loop to be restarted at the beginning. All of the loop variables are reinitialized. Typically, some side-effects will have been performed so that restarting the loop will lead to a different computation.

Meta Sequence Functions

The meta sequence functions take in ordinary functions and convert them into sequence functions. Each one takes in one or more functional arguments. Each of these can be either a quoted function name, or a quoted lambda expression, or a macro which expands into either one.

mapS *function sequence ...*

The nth element of the output sequence is computed by applying *function* to the nth elements of the input sequences. However, if the nth element of any of the input sequences is empty then *function* is not applied and the nth element of the output is empty. Note that the length of the output sequence is the same as the length of the shortest input sequence.

e.g., `(mapS #'+ [1 _ 2 3 4] [1 2 _ 3]) => [2 _ _ 6]`

scanS *function init sequence ...*

This is just like MAPS except that it has an internal state variable. The initial (zeroth) value of this variable is the unitary value *init*. The elements of the output are the successive values of the state not including its zeroth value. The nth value of the state is computed by calling *function* with the prior value of the state as its first argument and the nth elements of the inputs as its remaining arguments. However, if the nth element of any of the input sequences is empty then *function* is not applied, the state is not changed, and the nth element of the output is empty. The length of the output sequence is the same as the length of the shortest input sequence.

e.g., `(scanS #'+ 0 [1 _ 2 3 4]) => [1 _ 3 6 10]`

filterS *function sequence ...*

The elements of the output sequence are computed as follows. If the result of applying *function* to the nth elements of the input sequences is non-NIL then the nth element of the *first* input is used as the nth element of the output; otherwise the nth output element is empty. However, if the nth element of any of the input sequences is empty then *function* is not applied and the nth element of the output is empty. Note that the output sequence is exactly the same length as the shortest input sequence; however, some of the output sequence slots may be empty.

e.g., `(filterS #'> [1 _ 2 3 4] [0 2 _ 3]) => [1 _ _ _]`

reduceS *function init sequence ...*

This creates a sequence function with an internal state variable. The state is initialized to the (unitary) value *init*. The nth value of the state is computed by calling *function* with the prior value of the state as its first argument and the nth elements of the inputs as its remaining arguments. However, if the nth element of any of the input sequences is empty then *function* is not applied and the state is not changed. When the input sequences are exhausted, the final value of the state variable is returned as the (unitary) result. If there are no non-empty elements in the input sequences then the value *init* will be returned. This form is equivalent to: `(RLAST (SCANS` *function init sequence ...*`)` *init*`)`.

e.g., `(reduceS #'+ 0 [1 2 _ 3] [1 _ 2 3 4]) => 8`

e.g., `(reduceS #'+ 0 [] [1 _ 2 3 4]) => 0`

generateS *function init sequence ...*

This uses an internal state variable in order to generate a potentially infinite sequence of values. The unitary value *init* specifies the initial (first) value of the state. On the nth cycle of the loop, *function* is called with the nth value of the state as its first argument and the nth elements of the input sequences (if any) as its remaining arguments in order to compute the next value of the state. However, if the nth

element of any of the input sequences is empty then *function* is not called and the value of the state is not changed. The output sequence consists of all of the values of the state including the first one *init*. If there are no input sequences (the normal case) or if none of them are finite, then the output will be infinite. If any of the input sequences is finite, then the length of the output will be the same the length of the shortest input. Note that in this case, the final value of the state will not be returned as part of the output.

e.g., `(generateS #'1+ 0) => [0 1 2 3 4 5 6 7 ...]`

e.g., `(generateS #'(lambda (prev new) new) NIL [1 2 3 4]) => [NIL 1 2 3]`

truncateS *function sequence ...*

This is used to create sequence functions which take in potentially infinite sequences and return sequences which have been truncated to finite length. The *function* argument is applied to successive groups of corresponding elements of the input sequences. The output sequence is composed of the elements of the *first* input sequence up to but not including the first element corresponding to a non-`NIL` evaluation of *function*. As with the other meta sequence functions, if any of the nth elements of the input sequences are empty then *function* is not applied and the nth output element is empty. Note that the output sequence is typically shorter than any of the input sequences, and can be of length zero.

e.g., `(truncateS #'< [1 _ 2 3 4] [0 2 _ 4]) => [1 _ _]`

e.g., `(truncateS #'> [1 _ 2 3 4] [0 2 _ 4]) => []`

enumerateS *truncate-function generate-function init*

This is an abbreviation for (`TRUNCATES` *truncate-function* (`GENERATES` *generate-function init*)). It is the preferred way to define an enumerator.

e.g. `(enumerateS #'zerop #'1- 5) => [5 4 3 2 1]`

at-start *function arg ...*

This computes (*function arg ...*) in the initialization code before a loop begins. All of the *args* must be unitary values.

at-end *function arg ...*

This computes (*function arg ...*) in the epilog code after a loop ends. All of the *args* must be unitary values. They can be values returned by reducers. Note that this will not be executed if the loop is terminated via a `DONE` with arguments or by some extraordinary exit such as a `THROW`.

at-unwind *function arg ...*

This computes (*function arg ...*) in an `UNWIND-PROTECT` wrapped around the loop. All of the *args* must be unitary values. They can be values returned by reducers. The difference between this and `AT-END` is that it will be executed no matter how the loop is terminated.

Predefined Generators

Gsequence *arg*

This takes in a unitary argument and produces an infinite sequence of that value. Note that the successive elements of the sequence will all be EQ.

e.g., (Gsequence 1) => [1 1 1 ...]

Gprevious *sequence* **&optional** (*first* **NIL**)

This takes in a sequence and returns a sequence which is shifted right one position. *First* is used as the first element of the output, and the last element of the input is discarded.

e.g., (Gprevious [1 2 3 4] 0) => [0 1 2 3]

Glist *list*

This generates the successive elements of *list*. It will get an error if it encounters a non-list CDR.

e.g., (Glist '(1 2 3)) => [1 2 3 NIL NIL NIL ...]

Gsublists *list*

This generates the successive CDRs of *list*. It will get an error if it encounters a non-list CDR.

e.g., (Gsublists '(1 2 3)) => [(1 2 3) (2 3) (3) NIL NIL NIL ...]

Grange &optional (*first* 1) (*step-size* 1)

This generates fixnums from *first* adding *step-size* at each step. Note that *step-size* can be negative.

e.g., (Grange 10 2) => [10 12 14 ...]

Predefined Enumerators

Elist *list*

This enumerates the successive elements of *list* up to and not including the first NULL sublist. It will get an error if it encounters a non-list CDR.

e.g., (Elist '(1 2 3)) => [1 2 3]

e.g., (Elist nil) => []

Esublists *list*

This enumerates the successive CDRs of *list* up to and not including the first NULL sublist. It will get an error if it encounters a non-list CDR.

e.g., (Esublists '(1 2 3)) => [(1 2 3) (2 3) (3)]

Elist* *list*

This enumerates the successive elements of *list* up to and including the first NULL or non-list sublist.

e.g., (Elist* '(1 2 . 3)) => [1 2 3]

e.g., (Elist* NIL) => [NIL]

Eplist *plist* => *sequence-of-properties sequence-of-values*

This creates two sequence outputs consisting of the successive property names and property values respectively of the naked plist *plist*. Note that the function PLIST returns the CDR of a naked plist, not a naked plist.

e.g., (Eplist '(NIL A 1 B 2)) => [A B] [1 2]

Ealist *alist* => *sequence-of-keys sequence-of-values*

This creates two sequences as outputs consisting of the successive keys and values respectively of *alist*. It requires that the lists of values associated with each key be lists. They may have 0, 1, or more values in them.

e.g., `(Ealist '((A 1) (B) (C 2 3)))` => `[A C C] [1 2 3]`

Erange *first last* &optional (*step-size* 1)

Creates a sequence of integers by counting from *first* to *last* by the positive increment *step-size*.

e.g., `(Erange 4 8 2)` => `[4 6 8]`

Evector *vector* &optional (*first* 0) (*last* `(1- (array-length vector))`)

This enumerates the successive elements of a one dimensional array. You can specify a subrange of indices by specifying *first* and *last*. (Note that this will not work on MacLisp arrays of numeric type.)

e.g., `(Evector <1 2 3>)` => `[1 2 3]`

Efile *file-name*

This creates a sequence by doing successive reads on the file until end of file is reached. *File-name* can be anything acceptable to **OPEN**.

e.g., `(Efile "data.lisp")` => `[1 2 3]`
if the file contains "1 2 3"

Predefined Filters

Fgreater *sequence* &optional (*limit* 0)

This takes in a sequence of fixnums and restricts it to a sequence containing only elements greater than *limit*.

e.g., `(Fgreater [1 2 3] 2)` => `[_ _ 3]`

Predefined Reducers

Rlast *sequence* &optional (*default* **NIL**)

This takes in a sequence and returns its last value. If the sequence has zero length then *default* is returned.

e.g., `(Rlast [1 2 3])` => `3`

e.g., `(Rlast [] NIL)` => **NIL**

Rignore *sequence*

This takes in a sequence and returns no values at all. It is useful in many of the same situations as **MAPC**.

e.g., `(Rignore [1 2 3])` =>

Rlist *sequence*

This creates a list of the elements in *sequence*. The order of the elements is preserved.

e.g., `(Rlist [1 2 3])` => `(1 2 3)`

e.g., `(Rlist [])` => **NIL**

Rbag *sequence*

This creates a list of the elements in *sequence*. The order of the elements in the list is undefined. This is more efficient if you really do not care what the order is. (The order ends up reversed, but you should not depend on that, because it could change at any time.)

e.g., `(Rbag [1 2 3]) => (3 2 1)`

Rlist* *sequence*

This creates a list of the elements in *sequence* with the last element of the *sequence* ending up as the CDR of the last CONS cell in the list.

e.g., `(Rlist* [1 2 3]) => (1 2 . 3)`

e.g., `(Rlist* [1]) => 1`

e.g., `(Rlist* []) => NIL`

Rnconc *sequence*

This creates a list by NCONCing together the successive elements of *sequence*. This is what MAPCAN does to create its output.

e.g., `(Rnconc [(1 2) NIL (3 4)]) => (1 2 3 4)`

Rappend *sequence*

This creates a list by APPENDing together the successive elements of *sequence*.

e.g., `(Rappend [(1 2) NIL (3 4)]) => (1 2 3 4)`

Rset *sequence*

This combines the elements in *sequence* into a list omitting any duplicate elements. The order of this list is undefined. The predicate which is used to test for duplicates is EQUAL.

e.g., `(Rset [1 1 (2) (2)]) => ((2) 1)`

Reqset *sequence*

This is the same as RSET except that the test for duplicates is EQ instead of EQUAL.

e.g., `(Reqset [1 1 (2) (2)]) => ((2) (2) 1)`

Rplist *sequence-of-properties sequence-of-values*

This takes in a sequence of property names, and a sequence of values and creates a naked plist. Note that the function SETPLIST expects to receive the CDR of a naked plist as its second argument.

e.g., `(Rplist [A B] [1 2]) => (NIL B 2 A 1)`

Ralist *sequence-of-keys sequence-of-values*

This takes in a sequence of keys, and a sequence of values and creates an alist. All of the values which have the same key are combined into a single entry in the alist headed by the key. The predicate which is used to test for equality of keys is EQUAL.

e.g., `(Ralist [(A) B (A) B] [1 2 3 4]) => ((B 4 2) ((A) 3 1))`

Reqalist *sequence-of-keys sequence-of-values*

This is identical to RALIST except that the test for key equality is EQ.

e.g., `(Reqalist [(A) B (A) B] [1 2 3 4]) => (((A) 3) (B 4 2) ((A) 1))`

Rvector *vector sequence* **&optional** (*first* 0) (*last* (1- (array-length vector)))

This takes in a one dimensional array and a sequence of elements and stores those elements in successive positions in the array. You can specify a specific subrange in the array. (This will not work with MacLisp arrays of numeric type.) Note that this reducer is unusual in that it contains a terminator and will stop the loop as soon as the vector is full.

e.g., (Rvector <NIL NIL NIL NIL> [1 2 3]) => <1 2 3 NIL>

e.g., (Rvector <NIL NIL> [1 2 3]) => <1 2>

Rfile *file-name sequence*

This takes in a sequence and writes all of its elements into a file. *File-name* can be anything acceptable to OPEN.

e.g., (Rfile "data.lisp" [1 2 3]) => T

"<cr>1 <cr>2 <cr>3 " is printed in "data.lisp"

Rsum *sequence-of-integers*

Computes the sum of the integers in its input.

e.g., (Rsum [1 2 3]) => 6

Rsum$ *sequence-of-flonums*

Computes the sum of the flonums in its input.

e.g., (Rsum$ [1.1 2.2 3.3]) => 6.6

Rmax *sequence-of-numbers*

Computes the maximum of the numbers in its input. Returns NIL if the input has length zero.

e.g., (Rmax [1 2 3]) => 3

e.g., (Rmax []) => NIL

Rmin *sequence-of-numbers*

Computes the minimum of the numbers in its input. Returns NIL if the input has length zero.

e.g., (Rmin [1 2 3]) => 1

e.g., (Rmin []) => NIL

Rcount *sequence*

Computes the number of elements in its input.

e.g., (Rcount [1 2 3]) => 3

Rand *sequence*

Computes the AND of all of the elements of *sequence*. As with AND, the return value is either NIL or the last element of the input.

e.g., (Rand [1 2 3]) => 3

e.g., (Rand [1 NIL 2]) => NIL

e.g., (Rand []) => T

Rand-fast *sequence*

This is the same as RAND except that the loop is terminated as soon as a NIL value (if any) is encountered.

e.g., (Rand-fast [1 2 3]) => 3

uence

Computes the OR of all of the elements of *sequence*. As with OR, the return value is either NIL or the first non-NIL element of the input.

e.g., (Ror [1 2 3]) => 1

e.g., (Ror [NIL NIL]) => NIL

e.g., (Ror []) => NIL

ast *sequence*

This is the same as ROR except that the loop is terminated as soon as a non-NIL value (if any) is encountered.

e.g., (Ror-fast [1 2 3]) => 1